Dipartimento di Informatica
Università di L'Aquila
Via Vetoio, I-67100 L'Aquila, Italy
http://www.di.univaq.it

Tesi di Dottorato

Model Checking for the Analysis and Control of Complex and Non-deterministic Systems

Fabio Mercorio

Gennaio 2012

Settore Scientifico Disciplinare INF/01
XXIV Ciclo

Tutore	Coordinatore del Dottorato
Prof. Giuseppe Della Penna	Prof. Stefania Costantini

Dipartimento di Informatica
Università di L'Aquila
Via Vetoio, I-67100 L'Aquila, Italy

http://www.di.univaq.it

Ph.D. Thesis

Model Checking for the Analysis and Control of Complex and Non-deterministic Systems

Fabio Mercorio

January 2012

Advisor
Prof. Giuseppe Della Penna

PhD Program Supervisor
Prof. Stefania Costantini

Al mio papà: ora so cosa significa ☺

To my dad: now I know what it means ☺

ABSTRACT

In the last two decades, the use of intelligent planning algorithms, complex controllers, and automated verification processes is growing apace, having a great impact in many industrial fields, as robotics, manufacturing processes, and embedded systems, which now are present in an increasing number of everyday products and appliances. Moreover, many processes take place in an environment having variable and unpredictable influences on the system dynamics, making the problem of dealing with these *non-deterministic* behaviours a very significant concern. As a result, a growing synergy between Control and AI Planning communities has been established, with the aim to develop algorithms and tools able to cope with such systems. In particular, it is interesting to evaluate *robustness*, verify the *correctness* and compute *plans* to execute activities. To this aim, *formal methods* (and in particular *model checking*) are well-suited to deal with these issues.

For several years, both control and planning problems have been addressed only through *symbolic* model checking, which has been successfully applied to a wide class of systems. Nevertheless, there are still some open issues in dealing with Discrete Time Hybrid Systems (DTHS), whose state description involves both continuous and discrete variables, as well as systems with a complex nonlinear dynamics, for which symbolic approaches are hard to apply. To this regard, we focus on the use of *explicit* model checking, which is based on the explicit enumeration of the system states, to deal with control and planning problems in both deterministic and non-deterministic domains.

Nevertheless, the explicit approach is strongly affected by the so called *state explosion* problem. In order to mitigate this problem, a first contribution is the developing of a disk-based algorithm for the UPMurphi tool: a universal planner for continuous domains built on top of the Murphi model checker. We exploit the use of disk-based approach to analyse and control systems having a huge state space, showing a number of benchmarks and real world planning and control case studies. Moreover, we extend the use model checking to *database data quality* problems, using formal methods for the verification of *data consistency* defined over a set of data items, and evaluating the results on a real application of a Public Administration database provided by the C.R.I.S.P. research center.

Finally, we tackle with *non-deterministic systems* in which an action may have different outcomes, unpredictable at planning time, addressing the problem to synthesise a plan able to reach a goal in spite of the non-determinism, i.e., *strong* plan. Many approaches have been applied in literature, mainly based on *symbolic* model checking. As a novel contribution, we present an algorithm able to synthesise strong plans (if any) with *minimum cost* with respect to a given cost function (that is minimising the non-deterministic worst-case execution), analysing its complexity, correctness and completeness. Finally, we describe the implementation of the algorithm into UPMurphi and we test it on two continuous non-deterministic case studies.

ACKNOWLEDGMENTS

First of all, I want to give thanks to my wife Valentina, which lovingly tolerate me (I know that's hard), and to our son Pietro: you are the greatest contributor of this work, thanks to kept me awake (by night) writing the thesis....I'll never forget it.

My heartfelt thanks to my parents and my sister, which always encouraged every decision I made.

I am so grateful to my advisor, prof. Giuseppe Della Penna. Thanks for your guidance during these wonderful years and, mainly, thanks for your patience with me.

A special thank is reserved to Daniele for his unfailing support, especially when I wouldn't have wanted to be supported at all.

Warm thanks to all the researchers with whom I had the great pleasure to work: Benedetto, Enrico, Mario, Mirko and Roberto.

Finally, I thank my thesis reviewers, Prof. Jean-Marc Faure and Prof. Ganesh Gopalakrishnan for their careful reading and for their valuable comments and suggestions on how to continue and improve this work.

Fabio

“E si’ come essere suole che l’uomo va cercando argento e fuori de la ’ntenzione truova oro.”

Dante Alighieri. Convivio,II,xii

“And as it is wont to chance that a man goeth in search of silver and beyond his purpose findeth gold.”

Dante Alighieri. The Convivium,II,xii

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In the recent years, the magnitude of Information and Communication Technology (ICT) systems has grown apace, making these systems more and more complex with respect to their functionalities and purposes. Indeed, a great number of everyday products are the result of research in fields as robotics, manufacturing processes, communication technology, and embedded systems, which make a large use of intelligent planning algorithms as well as complex controllers and automated verification processes. Moreover, many systems present a non-deterministic behaviour, mostly due to unpredictable environmental conditions, making the problem of dealing with such domains a challenging task.

To this regard, a common effort between Control and AI Planning communities is in existence towards the developing of algorithms and tools able to *analyse* and *control* systems dynamics, focusing on the evaluation of their *robustness*, verification of *correctness* and automatic synthesis of *plans* to execute activities.

To cope with this task, one can perform *formal verification* as well as *automatically generate* correct reactive programs directly from the plant specification. To this aim, *formal methods* (and in particular *model checking*) are well-suited to deal with both issues.

1.1 MODEL CHECKING TECHNIQUES

Much time and efforts have been spent to develop techniques to support the software and hardware verification. To this regard, formal methods widely use mathematics for modelling and analysing of ICT systems, with the aim to establish, with mathematical rigor, the correctness of a suitably modelled system. It is worth noting that, any system verification which uses a model-based technique is only as good as the model of the system.

Roughly speaking, Model Checking [39, 108, 30, 86, 93, 10] (first introduced in [39]) is a brute-force hardware/software verification technique that explores the system behaviour (by checking all its possible moves) looking for an inconsistency with respect to the given semantics (i.e., an *error*). If the model checker (which is the software that implements a model checking technique) does not find any error, then the model of the system meets the designers requirements.

In other words, a model checker performs an exhaustive search in the system state-space looking for an error. If an error exists then the model checker returns the *error path* which leads the system from the initial configuration to the error one, providing *how* the system has reached the error.

In the last years, many model checkers have been developed (see e.g., SPIN [94], UPPAAL [118], NuSMV [139]). In particular, we can mainly distinguish between two kinds of model checking paradigms: *symbolic* (see, e.g., [8, 30]) and *explicit* (see, e.g. [68, 69]). The main difference between them is in the state space representation: the former uses a symbolic (usually compressed) representation (based on OBDDs, see Section 4.3.1 for details) which is successfully applied on discrete systems, whilst the latter works well on Discrete Time Hybrid Systems (DTHS), whose state description involves both continuous and discrete variables, also having a nonlinear dynamics. Although this approach is being successfully applied to a wide class of systems, there are still some open issues in dealing with systems having a complex nonlinear dynamics. Indeed, for this class of systems, current methodologies based on symbolic model checking as well as on dynamic programming are hard to apply. To this end, we focus on the use of *explicit* model checking, which is based on the explicit enumeration of the system states, to deal with both control and planning problems.

1.2 Explicit Model Checking in Deterministic Systems

The first part of the Thesis is devoted to the *planning*, *control* and *analysis* of *deterministic systems* via explicit model checking techniques, by focusing on systems which have a complex (also nonlinear) dynamics.

Typically *planning* concerns the problem of generating a sequence of actions, in order to move the system from a specified initial state to a desired goal state. The generation (synthesis) of a plan is generally performed at run-time on request. Differently, a *universal plan* [150] (or *controller*) is computed off-line, by exploring the system dynamics and synthesising a plan *for each* system state reachable from the initial ones. It is worth noting that, in some contexts, the meaning of universal plan is related to non-determinism in refer to the uncertainty about the initial states of the system, not about its dynamics, which could be deterministic. Nevertheless, a universal planner can be seen as a controller which summarises the commands (actions) to send to the plant in order to reach the goal, avoiding to spend time in the synthesis of real-time solutions. Indeed, many systems have low computational resources or require a small reactive time to perform actions. In these settings, the synthesis of a *universal plan* can represent a well-suited choice. Similarly to the model checking approach, the synthesis of plans requires to *explore* the system dynamics and to *evaluate* when the goal condition is reached. One can exploit these similarities between planning and model checking approaches, using a model checker like a planner by considering the goal condition as an *error* condition for the system. As a result, the *error trace* (if any) given by the model checker represents a *plan* for the planning problem.

To this regard, despite the idea to perform planning via model checking is not new [36, 82, 83, 143, 45], we propose to use the explicit model checking approach which works well on Discrete Time Hybrid Systems with a very complex dynamics (e.g., nonlinear). Indeed, in the literature, a growing number of motivating applications shows the importance of dealing with mixed discrete continuous domains [9, 90, 16, 145] as well as to define languages to describe them [129, 76, 80]. Hence, many approaches have been used to cope with such systems (e.g., Dynamic Programming [20], Flow Tubes [119, 122], MINLP [84]), also based on Symbolic Model Checking [65, 66].

However, if by one side the explicit approach is promising in verification of hybrid systems, on the other side it is strongly affected by the so called *state explosion* problem. In order to mitigate this problem, a first contribution of this thesis is the developing of a disk-based algorithm for the UPMurphi tool: a universal planner for continuous domains built on top of the Murphi model checker. We exploit the use of disk-based approach, that makes it able to analyse and control systems having a huge state space, by using the disk during the synthesis process. Moreover, this improvement allows one to *pause* the synthesis process by storing the expanded graph of the dynamics on disk, and *resuming* the verification later, also on other machines. Then, we apply the disk-based approach to a number of benchmarks and real world planning and control case studies.
Furthermore, we also apply model checking to another class of problems, namely the *database data quality* problems. Data quality is a general concept and it can be characterised by many dimensions (e.g., *accuracy*, *consistency*, *accessibility*) [13]. In our context, we focus on *consistency*, a dimension which can be modelled and verified using formal methods. In particular, we propose a methodology which uses formal methods by looking for the violation of semantic rules (i.e., *inconsistencies*) defined over a set of data items. Then, as a first application, we evaluate the benefits given by our approach on a real industrial data quality case study of a Public Administration database, provided by the C.R.I.S.P. research center [43].

1.3 Explicit Model Checking in Non-Deterministic Systems

If in a deterministic system a plan reaches a goal in all its executions, in a non-deterministic one no guarantees are given about the influence that non-determinism has on plans executions. Roughly speaking, a non-deterministic system represents a particular form of uncertainty in which system's action may have different outcomes, unpredictable at planning time. Then, given an action, it is impossible for the planner to know a priori which the outcome will be. The concept of *strong* plan fills this gap by ensuring that a strong plan always reaches a goal regardless of the system non-determinism. For the sake of clarity, it is worth noting that strong plan and universal plan in a non-deterministic context are closely realted since both approaches aim to find a path to a goal starting from any state reachable from the initial ones. Furthermore, systems may have or not probabilities associated to the actions outcomes. We focus on systems in which no probabilities are given.

Many approaches have been applied to synthesise strong plans, mainly based on *symbolic* model checking [36, 115, 112, 111, 45] also using heuristics to prune the state space [78, 125, 106]. As a novel contribution, we present an algorithm able to synthesise strong plans with *minimum cost* with respect to a given cost function. The algorithm looks for a strong plan (if any) minimising the cost of the non-deterministic worst-case execution. We analyse the algorithm's complexity, proving its correctness and completeness. Finally, we describe the implementation of the cost-optimal strong planning algorithm into UPMurphi and we test it on two continuous non-deterministic case studies.

1.4 Thesis Structure

The Thesis is composed by three main Parts, organized as follows:

Part I introduces the theoretical basis, looking at explicit and symbolic model checking approaches in Chapters 3 and 4 respectively, closely look to the Murphi model checker, since all our implementations are derived from it. Then, Chapter 5 is devoted to other methodologies related to planning and model checking.

Part II discusses the use of explicit model checking to deal with *deterministic systems*. Chapter 6 introduces the UPMurphi planner and Chapter 7 presents the improvements of UPMurphi: namely the V-UPMurphi tool, whose application to some real-world case study is provided in Chapter 8. Finally, Chapter 9 shows how formal methods, and in particular model checking, can be applied to the problem of *data quality analysis* on dirty database, providing a first experimental results on a real case scenario.

Part III is devoted to the analysis of systems having a non-deterministic behaviour. In particular, Chapter 10 discusses the problem of synthesis of *strong* plans whilst Chapter 11 proposes a novel algorithm to synthesise *optimal* strong plans for non-deterministic systems. Finally, Chapter 12 figures out some case studies on which we applied the algorithm.

Finally, Chapter 13 contains concluding remarks and discussions about the future directions.

Part I

Theoretical Basis

In the first part, we introduce the notation and formal models we will use during the Thesis. In Chapter 2 we give an introduction about the formalisms used to model a system. Then, we closely look at model checking techniques in Chapters 3 and 4, discussing the explicit and symbolic approaches respectively, with a particular attention to the Murphi model checker, since all our implementations are derived from it.
Finally, Chapter 5 is devoted to other methodologies related to planning and model checking.

CHAPTER 2

SYSTEM MODELLING

In this section introduce the basis on formalisms used to describe the behaviour of a system (i.e., a *model*). In Section 2.1 we introduce hybrid automata which are used to model mixed-discrete continuous systems. Then in Sections 2.2 and 2.3 we focus on formalisms that allow one to describe systems having a finite number of states in both deterministic and non-deterministic cases. Finally, in Section 2.4 we briefly describe the language used to model planning domains.

2.1 HYBRID SYSTEMS

A dynamic system describes the evolution of each *state* of the system with respect to the *time*. Hence, in order to better introduce the concept of hydrid systems, we can characterise a dynamic system considering (1) its **state** and (2) the **time**. More precisely, we can distinguish between the following systems:

Continuous. When all n variables of the state belong to $\mathbb{R}^n$.

Discrete. When all n variables of the state belong to a finite set of values $Q = \{q_1, \ldots, q_k\}$.

Hybrid. Let $n = n_1 + n_2$ be all the variables of the state, then the first n_1 variables belong to $\mathbb{R}^{n_1}$ while the remaining n_2 variables take values in $Q = \{q_1, \ldots, q_k\}$.

We now closely look at the evolution of the state, i.e., how the state variables change along the time.

Continuous Time. When the time is a subset of $\mathbb{R}$, then the state evolution is described by an Ordinary Differential Equation.

Discrete Time. When the time is a subset of $\mathbb{Z}$, then the state evolution is described by a Finite Difference Equation.

Finally, the characteristic of the dynamics can further distinguish between **linear** and **nonlinear** systems.

2.1.1 Discrete Time Hybrid Systems

Generally speaking, a Hydrid System [161] is a formal model for mixed discrete-continuous systems. More precisely, a hybrid system is a kind of *dynamical system* in which the dynamics allows the presence of both continuous and discrete variables.

In other words, hybrid systems are ensembles of interacting *discrete* and *continuous* systems where the former operates on a discrete state and performs discontinuous state changes at discrete time points. Differently, the latter operates on a continuous state which evolves continuously.

To give an example, a car engine having a fuel injection (continuous) regulated by a micro-processor (discrete) represents a hybrid system. This simple example should be adequate to guess that, in the real life, many applications are hybrid systems. More formally, we give the following definition.

Definition 1. *A* Discrete Time Hybrid System *(DTHS) is a tuple $\mathcal{H}$ = (X,Q,U,W,I,f,p) where:*

- $X = \times_{i=1}^{n}[a_i,b_i]$, *with* $[a_i,b_i]$ *a bounded interval of the reals* $\mathbb{R}$.
- $Q = \times_{i=1}^{k}[c_i,d_i]$, *with* $[c_i,d_i]$ *a finite subset of the integers* $\mathbb{Z}$.
- $U = \times_{i=1}^{m}[\alpha_i,\beta_i]$, *with* $[\alpha_i,\beta_i]$ *a bounded interval of the reals* $\mathbb{R}$.
- $W = \times_{i=1}^{r}[\gamma_i,\mu_i]$, *with* $[\gamma_i,\mu_i]$ *a finite subset of the integers* $\mathbb{Z}$.
- *I is a subset of $X \times Q$.*
- *f is a function from $X \times Q \times U \times W$ to X s.t. for each $q \in Q$, $w \in W$, λxu $[f(x,q,u,w)]$ is a continuous function of (x,u) (where λ is the abstraction operator).*
- *p is a function from $X \times Q \times U \times W$ to Q.*

The state space of $\mathcal{H}$ is $S = X \times Q$. A *state* for $\mathcal{H}$ is a pair $s = (x, q)$ in S, where $x \in X$ and $q \in Q$.

A *run* for the DTHS $\mathcal{H}$ is a (finite or infinite) sequence of states and actions $(x(0),q(0),u(0),w(0)),\ldots,(x(t),q(t),u(t),w(t)),\ldots$ s.t. we have:

- $(x(0),q(0)) \in I$
- $x(t+1) = f(x(t),q(t),u(t),w(t))$ for each time t.
- $q(t+1) = p(x(t),q(t),u(t),w(t))$ for each time t.

If $\pi = (x(0), q(0), u(0), w(0))$, $(x(1), q(1), u(1), w(1))$, $\ldots$ is a run of $\mathcal{H}$ we denote with $\pi(t)$ the t-th state element of π. That is $\pi(t) = (x(t), q(t))$. Furthermore we write $\varphi(x(t), q(t), u(t), w(t))$ for $(f(x(t), q(t), u(t)), p(x(t), q(t), w(t)))$.

To convey to the reader the motivations behind our formalism, we make the following observations.

First we observe that $x \in X$ is the vector of the *continuous components of the state*, $q \in Q$ is the vector of the *discrete components of the state*, $u \in U$ is the vector of the *continuous components of the control actions*, and $w \in W$ is the vector of the *discrete components of the control actions*. I is the set of *initial states*.

Moreover, the function f assigns a region X_i in the *continuous state space* X and a dynamics which acts on the region X_i when the discrete state is q_i. Roughly speaking, to every discrete state corresponds a *mode* of the system.

The following example should help to clarify the matter.

Example (Water Tank) The two tank system, shown in Figure 2.1, consists of two tanks containing water. Both tanks are leaking at a constant rate. Water is added at a constant rate to the system through a hose, which at any point in time is dedicated to either one tank or the other. It is assumed that the hose can switch between the tanks instantaneously.

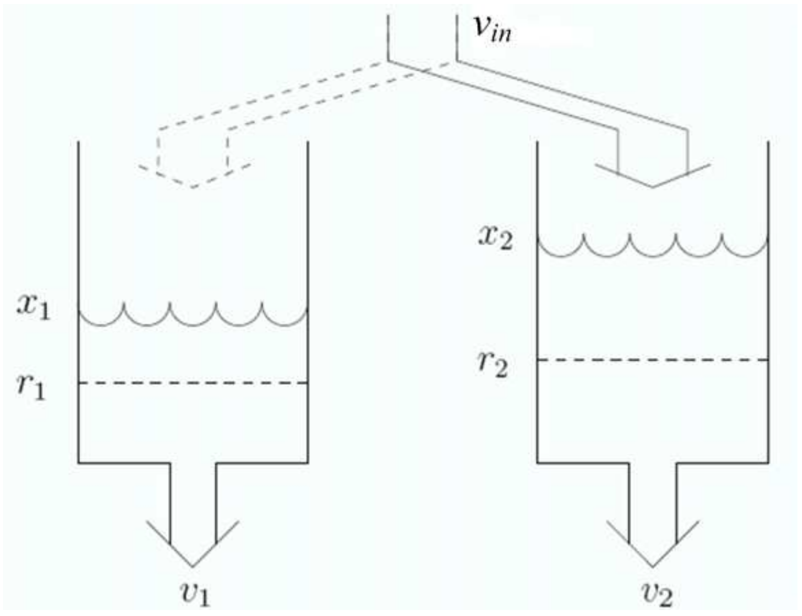


Figure 2.1: The water tank system

For $i \in \{1, 2\}$, let x_i denote the volume of water in Tank i and $v_i > 0$ denote the constant flow of water out of Tank i. Let v_{in} denote the constant flow of water into the system. The objective is to keep the water volumes above r_1 and r_2, respectively, assuming that the initial water volumes satisfy this constraint. A controller switches the inflow to Tank

1 whenever $x_1 \leq r_1$ and to Tank 2 whenever $x_2 \leq r_2$.

It is straightforward to define a discrete time hybrid system to describe this process:

- $X = [0, \mathtt{MAX\_}x_1] \times [0, \mathtt{MAX\_}x_2]$;
- $Q = \{q_1, q_2\}$;
- $U = \emptyset$;
- $W = [1,2]$;
- $I = Q \times \{x \in X | x_1 \geq r_1 \wedge x_2 \geq r_2\}$;
- $f(x,q,u,w) = \begin{cases} (x_1 + v_{in} - v_1, x_2 - v_2) & \text{if} \quad w = 1 \\ (x_1 - v_1, x_2 + v_{in} - v_2) & \text{if} \quad w = 2 \end{cases}$
- $p(x,q,u,w) = \begin{cases} q_1 & \text{if} \quad w = 1 \\ q_2 & \text{if} \quad w = 2 \end{cases}$

2.2 Finite State Systems

A Finite State System is a dynamic system which represents an abstract and discrete computational model often used to reproduce (and verify, as in the case of model checking) the behaviour of a given system.

Definition 2 (Finite State Systems)**.** *A* Finite State System *(*FSS*) $\mathcal{S}$ is a 4-tuple (S,I,A,F), where: S is a finite set of* states*, $I \subseteq S$ is a finite set of* initial states*, A is a finite set of* actions *and $F : S \times A \to S$ is the* transition function*.*

Definition 3 (Transition Function)**.** *Let $\mathcal{S}$ be a FSS, $s, s' \in S$ and $a \in A$, then there exists a transition function $F(s,a) = s'$ iff the state s can reach state s' via action a.*

Moreover, we denote with $F(s,a)$ the successor state of s through action a, i.e. the state s' s.t. $F(s,a,s') = 1$.

Definition 4 (Trajectory)**.** *A* trajectory *in the FSS $\mathcal{S} = (S,I,A,F)$ is a sequence $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \ldots a_{n-1} s_n$ where, $\forall i = 0, \ldots, n-1$, $s_i \in S$ is a state, $a_i \in A$ is an action and $F(s_i, a_i, s_{i+1}) = 1$. If π is a trajectory, we write $\pi_s(k)$ (resp. $\pi_a(k)$) to denote the state s_k (resp. the action a_k). Finally, we denote with $|\pi|$ the length of π, given by the number of actions.*

Definition 5 (Reachable States)**.** *Let $s_I \in I$ be an initial state of the FSS $\mathcal{S} = (S,I,A,F)$. Then, we say that a state s' is* reachable *from s_I iff there exists a trajectory π in $\mathcal{S}$ such that $\pi_s(0) = s_I$ and $\pi_s(k) = s'$ for some $k \geq 0$. We denote with* Reach(s) *the set of states reachable from s. Analogously, we denote with* $\mathrm{Reach}^{-1}(s)$ *the set of states from which it is possible to reach the state s, that is* $\mathrm{Reach}^{-1}(s) = \{s' \in S | s \in \mathrm{Reach}(s')\}$.

By abuse of notation, we denote as *Reach(S)* the set of reachable states for the system $\mathcal{S}$.

In order to perform the verification as well as the analysis task, the system should have a finite number of states. It is worth noting that, although this restriction that is quite theoretically relevant, its impact in the practice is limited since many systems can be modelled having a finite number of states.

Note that, if the system is a DTHS, given a suitable discretisation for sampling time and variables, we can easily obtain a Finite State Systems from a DTHS.

2.3 Non-Deterministic Finite State Systems

Definition 6 (Non-Deterministic Finite State System)**.** *A* Non-Deterministic Finite State System *(*NDFSS*)* $\mathcal{S}$ *is a 4-tuple (S,*s_0*,*$\mathcal{A}$*,F), where: S is a finite set of* states*,* $s_0 \in S$ *is the* initial state*,* $\mathcal{A}$ *is a finite set of* actions *and* $F : S \times \mathcal{A} \to 2^S$ *is the* non-deterministic transition function*, that is* $F(s,a)$ *returns the set of states that can be reached from state s via action a.*

It is worth noting that we are restricting our attention to NDFSS having a single initial state s_0 only for the sake of simplicity. Indeed, if we give a NDFSS $\mathcal{S}'$ with a set of initial states $I \subseteq S$, we may simply turn it into an equivalent NDFSS by adding a dummy initial state connected to all the states in I by a deterministic transition with fixed cost.

The non-deterministic transition function implicitly defines a set of transitions between states which, in turn, give raise to a set of trajectories as specified in the following definitions.

Definition 7 (Non-Deterministic Transition)**.** *Let* $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$ *be an* NDFSS*. A* non-deterministic transition τ *is a triple of the form* $(s,a,F(s,a))$ *where* $s \in S$ *and* $a \in \mathcal{A}$*. A* deterministic transition *(or simply a* transition*)* τ *is a triple of the form* (s,a,s') *where* $s,s' \in S$*,* $a \in \mathcal{A}$ *and* $s' \in F(s,a)$*. We say that* $\tau = (s,a,s')$ is in $(s,a,F(s,a))$ *if* $s' \in F(s,a)$*. We denote with* $\mathcal{S}_\tau$ *the set of all the transitions in* $\mathcal{S}$*.*

Definition 8 (Non-Deterministic Trajectory)**.** *A* trajectory π *from a state s to a state* s' *is a sequence of transitions* $\tau_0, \ldots, \tau_n$ *such that:*

- τ_0 *has the form* (s,a,s_1) *for some* s_1 *and some a,*
- τ_n *has the form* (s_n, a', s') *for some* s_n *and some* a'*,*
- $\forall i = 0, \ldots, n-1$*, if* $\tau_i = (s_i, a_i, s_{i+1})$ *for some* s_i, a_i, s_{i+1}*, then* $\tau_{i+1} = (s_{i+1}, a_{i+1}, s_{i+2})$ *for some* s_{i+2}, a_{i+1}*.*

We denote with $|\pi|$ *the length of* π*, given by the number of transitions in the trajectory.*

As usual we stipulate that *the empty set of transitions* is a trajectory from any state to itself.

Definition 9 (Extrated Transition)**.** *Let* $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$ *be an* NDFSS *and* Π *be a set of non-deterministic transitions. We say that a transition* $\tau = (s, a, s')$ *is* extracted from Π *if* $(s, a, F(s, a)) \in \Pi$ *and* (s, a, s') *is in* $(s, a, F(s, a))$*. Similarly, we say that a trajectory* $\pi = \tau_0, \ldots, \tau_n$ *is* extracted from Π *if,* $\forall i = 0 \ldots n, \tau_i = (s_i, a_i, s_i') \in (s_i, a_i, F(s_i, a_i))$ *and* $(s_i, a_i, F(s_i, a_i)) \in \Pi$*. Finally, we say that a state* $s \in \Pi$ *if there exists a transition* $\tau = (s, a, s')$ *extracted from* Π.

2.4 PDDL/PDDL+

Generally speaking, given a system modelled in some formalism, a *planning problem* consists of finding a sequence of actions (i.e. a *plan)* which guarantees to achieve the goal starting from a specified initial condition of the system. Hence, a *planner* is a software system able to synthesise a plan by taking as input (1) the domain description and (2) a goal description. One of the most common planning description languages is the PDDL [129]. The Planning Domain Definition Language (PDDL) was released in 1998, and has since then become the standard language for the planning community as well as for the AI International Planning Competitions [2]. The PDDL is an action-centered language having a LISP-like syntax and inspired by the STRIPS formulation of planning problems.

A PDDL definition consists of two parts: the *domain* and the *problem*. The former contains the domain predicates and actions as well as types and constants, whilst the latter describes a particular instance of the problem to solve. The reader can find a complete description of classical planning domains at [2].

In the last years, the planning community has developed many expressive extensions to the PDDL language. In particular, the PDDL2.1 [76] extends PDDL to include numeric expressions and durative actions (i.e., actions having effects depending on their duration). To give an example, let us to consider a pump `p` that fills a tank `t` at a given rate, then continuous effect is written in the following style:

```
(increase (volume ?t) (* #t (refuel_rate ?p)))
```

where `#t` represents the time over which the effect has been active. However, PDDL2.1 is limited to a discrete modelling of time (i.e., the only time points that can be identified in a plan are those associated with the start/end points of actions selected by the planner). Hence, in order to allow the PDDL to represent mixed discrete-continuous domains, in 2001 Fox and Long introduced the PDDL+ [77], providing a formal mapping between PDDL+ and Hybrid Automata and showing that the PDDL+ is strictly more expressive than PDDL2.1. In synthesis, PDDL+ introduces two important features with respect to PDDL2.1:

Process: It is a construct able to model continuous change in the world by modifying numeric values continuously. The continuous change on a system variable is activated by the process whenever its precondition is satisfied.

Event: An exogenous event is used to describe instantaneous changes in the world that may occur as a consequence of change, not necessarily a direct consequence of the actions of an executive. Differently from a process, an event is instantaneous and may affect only discrete variables.

Note that both processes and events have effects on the system behaviour in spite of the actions selected by the planner since they do not form part of the plan.

An example, as given in [77] should clarify the matter. A car that needs to cover a specified distance d in the least possible time. To change the car velocity v, we can accelerate or decelerate so incrementing or decrementing, respectively, the current acceleration by $1m/s^2$. Moreover, if the velocity is greater than a given threshold k, the wind resistance will start to slow the vehicle. Thus, the velocity of the vehicle is governed by the two following differential equations, according to whether $v < k$ or $v \geq k$:

$$\begin{array}{lcll} \frac{dv}{dt} & = & a & \text{if } v < k \\ \frac{dv}{dt} & = & a - 0.1(v-k)^2 & \text{if } v \geq k \end{array} \tag{2.1}$$

Finally, as a further constraint, the engine explodes if the velocity is greater than a maximum threshold E. The PDDL+ domain and problem are given in Figure 2.2. Note that actions *accelerate*,*decelerate* have an instantaneous effect (i.e., they increase/decrease the actual acceleration) whilst concurrent processes *moving* and *windResistance* have effect on the variable v. Then, *engineExplode* is an exogenous event that models the engine explosion when $v > k$.

```
(define (domain car)
(:requirements :fluents :time :negative-preconditions)
(:predicates (running) (stopped) (engineBlown))
(:functions (d) (v) (a) (k) (E))

(:process moving
:parameters ()
:precondition (and (running))
:effect (and (increase (v) (* #t (a))) (increase (d) (* #t (v)))))

(:action accelerate
  :parameters()
  :precondition (and(running) )
  :effect (and (running) (increase a 1) ))

(:action decelerate
  :parameters()
  :precondition (and(running))
  :effect (and (running)(decrease a 1)))

(:process windResistance
  :parameters ()
  :precondition (and (running) (>= (v) k))
  :effect (decrease (v) (* #t (* 0.1 (* (- (v) k) (- (v) k))))))

(:event engineExplode
  :parameters ()
  :precondition (and (running) (>= (a) 1) (>= (v) E))
  :effect (and (not (running)) (engineBlown) (assign (a) 0))))

(define (problem car)
  (:domain car)
  (:init (not (engineBlown)) (running) (= d 0) (= a 0) (= v 0)) (= k 100) (= E 200)
  (:goal and ((>= d 20) not(engineBlown)) )
  (:metric minimize(total-time)))
```

Figure 2.2: Files car_domain.pddl and car_problem.pddl

CHAPTER 3

EXPLICIT MODEL CHECKING

Explicit Model Checking is an automated technique that, given (1) a finite-state model of a dynamic system and (2) a formal property, exhaustively checks whether this property holds for each state of that model. Generally speaking, a *model checker* is a tool able to solve a model checking problem.

The main aspect of *explicit* model checking is that each state is represented as the collection of its variable values and each visited state is stored in RAM (namely in the hash table). This quickly fills up all the available computation resources, especially the memory. This problem is often addressed as the *state space explosion* (i.e., the number of states grow exponentially with respect to the time). As for the theoretical computational complexity, model checking is P-SPACE complete.

A schematic representation of the model checking process is depicted in Figure 3.1. Looking at the figure, we can describe the explicit model checking process identifying three different phases:

System Modelling. A representation of the system is realised by modelling its state variables and dynamics (i.e., the system's evolution). Moreover, the system properties which we are interested to verify are formally written. This phase is carried out through the model description language of the model checker at hand.

System Verification. This phase is the *running phase* which depends on the approach applied (i.e., symbolic or explicit one). The model checker performs an exhaustive search in the system state-space looking for an error (i.e., a state $s \in S$ such that s violates φ). More precisely, in the explicit approach this phase works as follows:

1. obtain the transition graph of the system S (a transition graph specifies how S may go from a state to another state);
2. compute the *reachable* states, starting from a given set of initial states (*reachability analysis*);
3. if no errors exist, then the property is *satisfied* on the system model. Otherwise, the model checker returns a *counterexample* (i.e., it provides *how* the system has reached the error) which leads the system from the initial configuration to the error one.

System Analysis. In this phase, the results obtained by the model checker are analysed. If the system model meets the property then the modeller can verify the next one (if any). On the contrary, if the property is violated then one can analyse the counterexample by means of simulation (i.e., a verification step by step), then the model as well as to system property is refined and the model checking procedure iterated.

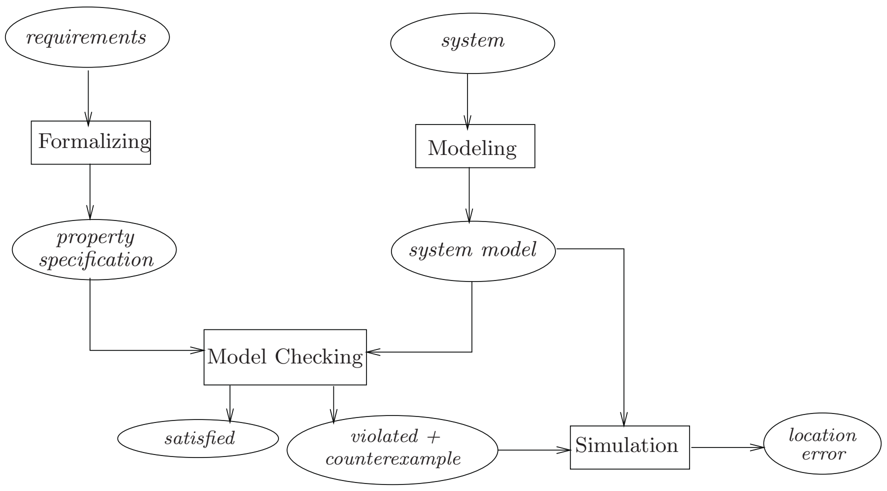


Figure 3.1: Schematic representation of the Model Checking approach (Taken from [10]).

The main obstruction for the verification via explicit model checking is in the reachability step. In fact, even if the formal description of S has a reasonable size, the number of states in S is exponential in the size of the description of S.

3.1 Model Checking on Finite State Systems

In order to verify *all the possible* states for which a system can be in, it should be required that the system has a *finite* number of states. For this reason, a system is often modelled as a FSS (according to Definition 2). Note that, whatever the system is modelled having a finite number of states, it may have an *infinite* number execution paths (i.e., the system can be in a deadlock state).

For the sake of completeness, we give the definition of Model Checking Problem on FSS.

Definition 10 (Model Checking Problem on FSS)**.** *Let $\mathcal{S} = (S, s_0, A, F)$ be an FSS. Let φ be a formula expressed in some formalism (i.e., the system specification). Then, a* model checking problem *(*MCP *in the following) is a triple $\mathcal{M} = (\mathcal{S}, \varphi, T)$ where $s_0 \in S$, and T is the finite temporal horizon.*

Then, a solution for $\mathcal{M}$ is a reachable trajectory π *(*plan*), according to Definition 4, $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \dots a_{n-1} s_n$ where: $\forall i \in 0, \dots, n-1$, $s_i \in Reach(S)$ and s_i satisfies φ whilst*

$s_n \in Reach(S)$ does not satisfy φ. If $\forall s \in Reach(S), s$ satisfies φ then the solution is an empty trajectory.

In the following we briefly describe the SPIN model checker, which actually represents one of the most used explicit model checker in the context of the verification of communication protocols and distributed software systems.

However, the tools developed as part of this Thesis are based on the CMurphi model checker, which is in turn based on Stanford's Murphi. Thus, in the last section of this Chapter, we will closely look at these verifiers and their input languages.

3.2 The SPIN Model Checker

SPIN (Simple Promela INterpreter) is an explicit model checker developed by Gerard J. Holzmann [94, 156] in 1997 for the verification of communications protocols, concurrent processes, witching systems, concurrent algorithms, railway signaling protocols etc. It has since become widely used for the verification of critical systems and protocols, for which it proved to be very effective, mainly in industries. SPIN uses the PROMELA (PROcess MEta LAnguage) specification language to describe input models, which is translated into a `C` program successively compiled and executed. A PROMELA model is a *finite-state* model, that is the state variables are always bounded variables (the boundedness is a guarantee for decidability). Note that, as said above, finite state models can still permit infinite executions. Then, the SPIN verification core works similarly to the Murphi verification algorithm (i.e., performing an exhaustive search in the state space). For the sake of brevity, we describe only the main characteristics of PROMELA language, for which the reader can found a complete description in [144].

Processes can be modelled explicitly by describing the behaviour of each process. Thanks to this, SPIN can automatically verify the system properties in all the possible interleaving processes execution. This feature is useful to verify *asynchronous* systems and communication protocols. To this aim, there are no global clocks as well as implied synchronisation between processes provided by PROMELA language.

Channels allow one to model message passing between processes (in Murphi it is possible to represent channels using arrays and applying the *symmetry reduction* to verify the system). Clearly, since the system to verify is a FSS, the channels should be bounded queues/buffers either buffered (asynchronously) or unbuffered (by synchronous rendezvous handshake between processes).

Safety and Liveness. As said above, a *safety* property states that something bad never happens in the system. Conversely, *liveness* property states that something good should eventually happen. Looking at the verification of such properties, the former is violated (and then it can be detected) in finite time whilst the latter requires to consider (at least some) infinite system execution to be checked. To this aim, SPIN can verify formulae expressed through Linear Temporal Logic, which allows one to model both safety and liveness properties, which are suffice to express any kind of system specification (as proved by Alpern and Schneider [5]).

In order to give an example, we show a simple PROMELA model which implements the well-known N-Peterson mutual exclusion algorithm, in which N processes must share the same resource without conflict, using only shared memory for communication between them.

The PROMELA code of 2-Peterson algorithm is shown in Figure 3.2. The process `proctype` describes the behaviour of the process. The statement `assert` is used by SPIN to verify the property in each process interleaving execution. Indeed, an assertion statement is always executable and has no effect on the state of the system when it is executed. Generally, it is used to model safety property (i.e., the mutual exclusion property in our example). Figure 3.3 show the verification output in which no deadlock or assert violation have been found.

```
/* Peterson s solution to the mutual exclusion problem - 1981 */

bool turn , flag [2];
byte ncrit ; /* critical section */
active [2] proctype user ()
{
  assert ( _pid == 0 || _pid == 1);
again :
  flag [ _pid ] = 1; /* processes communication via shared memory*/
  turn = _pid ;
  ( flag [1 - _pid ] == 0 || turn == 1 - _pid );
  ncrit ++;
  assert ( ncrit == 1); /* critical section */
  ncrit --;
  flag [ _pid ] = 0;
goto again
}

active proctype init(){
atomic(){
  run user(1);
  run user(2);
}
}
```

Figure 3.2: PROMELA code for the *N-Peterson* mutual-exclusion algorithm

Clearly, as for all the explicit model checkers, also SPIN is affected by the state explosion problem. To this regard, SPIN implements many state space reduction techniques to compress the state size (e.g., *hash compaction*) or the state space size (e.g., looking for symmetries with the *partial order reduction*).

```
spin -a  n-peterson
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c ./pan -m10000

(Spin Version 6.1.0 -- 4 May 2011)
 + Partial Order Reduction

Full statespace search for:
 never claim          - (not selected)
 assertion violations +
 cycle checks        - (disabled by -DSAFETY)
 invalid end states +

State-vector 20 byte, depth reached 24, errors: 0
       40 states, stored
       27 states, matched
       67 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

    2.539 memory usage (Mbyte)
```

Figure 3.3: SPIN execution output for the PROMELA model of Figure 3.2

3.3 The Murphi Verifier

Murphi [58, 59, 60, 137] is a formal verification tool for finite-state systems developed during the 1990's in the Stanford University Computer Systems Laboratory. The aim of the Murphi project was to show that formal verification tools could have practical value. Indeed, during the years it has been involved in the verification of many verification task, to give a few examples:

- Verification a cache coherence protocol in Stanford's DASH projects [121].
- Verification of the link-level protocol and coherence protocol in Sun's S3.mp multiprocessor [138].
- Verification of cryptographic protocols (Needham-Schroeder and TMN protocols) [132].
- Verification of SSL 3.0 protocols [133].
- Verification of SCI cache coherence protocol [157].

Moreover, many other tools based on Murphi have been developed, for a list of such tools see [136].

3.3.1 The Murphi Description Language

The Murphi description language [60] has been realised to be more simple as possible, supporting also non-deterministic behaviour. The system evolution is described through a set of iterated guarded commands, inspired to the UNITY modelling language. Since the Murphi language allows one to model a finite state system, now we briefly focus on how the elements of a FSS can be modelled in Murphi.

Let $\mathcal{S} = (S, I, A, F)$ be a FSS according to Definition 2 and let φ an invariant formula to be verified on $\mathcal{S}$, the model checking problem is modelled in Murphi language as follows:

Const, Type, Var. They describe the system state (i.e. $s \in S$) as a set of typed and bounded variables.

Startstate. It allows to describe all initial states $s_i \in I$ of the system from which the verification will start.

Guarded Rule. Each rule is a guarded command, consisting of an *action* (i.e., an $a \in A$) and a condition that allows the system to execute the transition from the current state s to the next one s' (i.e., it models a single transition $F(s, a, s')$). Hence, the *body* of each rule describes the behaviour of the action. In other words, the statement modifies the variables values.

Ruleset. It allows one to make each rule *parametric* with respect to a given variable, that can have any value in its bounded range. It can be also used to model *non-determinism* for the respective action.

Invariant. It uses the first order logic to model the invariant condition φ which will be satisfied in each state. In particular, we can use the boolean logic to describe *safety* formulae (i.e., something bad never happens).

3.3.2 The Murphi Verification Algorithm

It is now clear that a Murphi model is composed by a *statical* part (which models states, invariants and actions) and a *behavioural* part, which is able to describe the system evolution (state transitions). The Murphi "control structure“ performs a single infinite loop that, starting from any initial state, repeatedly executes three steps: (1) evaluate all the action guards, given the current values of the global variables, (2) choose one of the action whose guard is true and execute it, updating the variables. Then, (3) verify if the invariant condition φ is satisfied in the new obtained state.

It is worth to highlight the following:

1. The Murphi state-space exploration policy expands the system dynamics (i.e., the transition graph) through the *reachability analysis*, namely it visits (1) only the states reachable from the initial ones and (2) it visits a state only once. In general, the graph visit algorithm can be anyone, in particular Murphi allows the user to choose between Breadth First (BF) and Depth First (DF) visits . For example, the automatic verifier SPIN [156] uses the DF visit as default.

2. Since $\mathcal{S}$ is a FSS, the algorithm always terminates because it never visits the same state more than once.

3. The Murphi's data structure si composed by a Queue (Stack) when it performs a BF (DF) visit and a Hash Table used to store visited states, crucial to recognise visited states.

4. During the visit, if an error state which violates φ is found, then Murphi returns the *error trace* (i.e., a path from the initial state to the error one).

For the sake of completeness, the Algorithm 1 shows the pseudocode of the BF Murphi's visit.

Procedure 1 BFS($\mathcal{S}, \varphi$)

```
let $\mathcal{S} \leftarrow (S, I, A, F)$;
let $\varphi$ be an invariant condition;
$Q_S$ a FIFO queue;
$HT$ a hash table;
for all $s \in I$ do
  if $\varphi \not\models s$ then
    return $false$ // initial state does not satisfy the invariant
  end if
  Enqueue($Q_S$, $s$); // store $s$ in the queue, it will be expanded during the search
  Insert($HT$, $s$); // store $s$ as visited
  while ( $Q_S \neq \emptyset$) do
    $s \leftarrow$ Dequeue($Q_S$)
    for all $s' \in \{F(s,a) \mid a \in A\}$ do
      if ($s' \not\models \varphi$) then
        return $false$;
      end if
      if ($s' \notin HT$) then
        Insert($HT$, $s'$); // store it in $HT$
        Enqueue($Q_S$, $s'$); // and insert it in the queue
      end if
    end for
  end while
end for
return $true$; // all $s \in Reach(S)$ satisfy $\varphi$
```

3.3.3 A Toy Example

A small toy example could be useful to clarify the matter. Consider the *Discrete Time System* (DTS) defined by equation 3.1, where $\mathbf{x}(t)$ is the state value at time t and $\mathbf{d}(t) \in \{0, 1, 2\}$ is the disturbance value at time t.

$$\mathbf{x}(t+1) = \begin{cases} \mathbf{x}(t) + \mathbf{d}(t) & \text{if } \mathbf{x}(t) \leq 3 \\ \mathbf{x}(t) - \mathbf{d}(t) & \text{otherwise} \end{cases} \qquad \forall t, \quad \mathbf{x}(0) = 0. \tag{3.1}$$

Figure 3.4 shows the FSS corresponding to the DTS defined by Equation 3.1. The initial state **x**(0)=0 is shown with an ingoing arrow. Moreover, nodes are labelled with state values whilst edges are labelled with action values (which represent disturbances, in our case).

The Murphi code for the DTS in Equation 3.1 is given in Figure 3.5 where we have examples of the syntax as well as the language constructs described before.

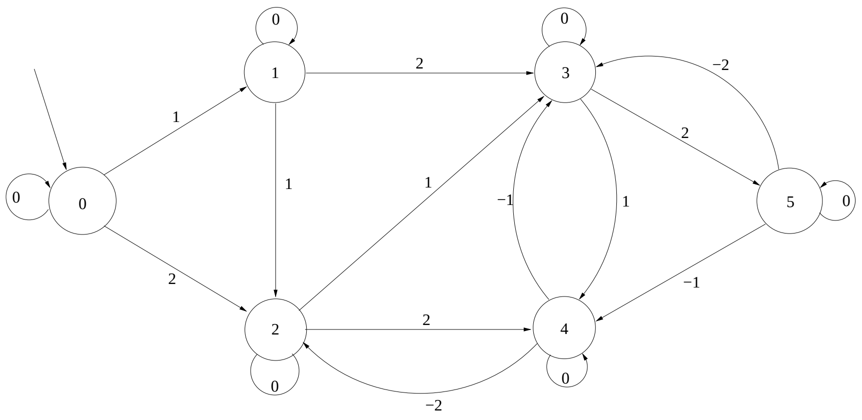


Figure 3.4: FSS for the discrete time system in Equation 3.1

We ran Murphi of the model in Figure 3.5 and we obtained the results summarised in Figure 3.6. Murphi returns an *error trace*, i.e. a (loopless) path in the graph in Figure 3.4 from an initial state to a state which violates the invariant property. Note that, replacing the $<$ sign in the invariant of Figure 3.5 with $\leq$ then the invariant property is always satisfied since all reachable states of the DTS defined by Equation 3.1 have a value less than or equal to 5 (see Figure 3.4).

Remark 3.3.1. *In the BF Algorithm 1 only reachable states are visited and thus stored in the hash table* `T`*. Hence the set of reachable states depends only on the* system dynamics. *For example, the set of reachable states for the FSS defined in Figure 3.4 is* $\{0,\ldots,5\}$. *This set does not depend on* `state_type` *(the type of variable x in Figure 3.5) as long as* `state_type` *contains* $\{0,\ldots,5\}$. *For example if in Figure 3.5 we change* `state_type` *declaration to* `state_type :  0..100` *the set of reachable states is still* $\{0,\ldots,5\}$.

3.4 The CMurphi Verifier

CMurphi [32] is built on top of the Murphi verifier. It provides three new kind of extensions:

```
/* constant declarations */
const
 MAX_STATE_VALUE : 5;
 MAX_DISTURB : 2;

 /* type declarations */
 type
  /* integers from 0 to 10 */
  state_type : 0 .. 10;
  /* integers from 0 to 2 */
  disturbance_type : 0 .. MAX_DISTURB;

 /* (global) variable declarations */
 var
  /* x is a variable of type state_type */
  x : state_type;

 /* define next state function */
 function next(x: state_type; d: disturbance_type): state_type;
 begin
  if (x <= MAX_STATE_VALUE - MAX_DISTURB) then
   return (x+d);
  else
   return (x-d);
  endif
 end;

 /* define initial state */
 startstate "startstate"
  begin
   x := 0;
  end

 /* nondeterministic disturbances trigger system transitions */
 ruleset d : disturbance_type do
  /* define parametric transition rule */
  /* here, d varies in disturbance_type, thus there are MAX_DISTURB + 1
     variants of rule "time step" */
  rule "time step" true ==>
  begin
   x := next(x, d);
  end;
 end;

 /* define property to be verified */
 invariant "x is not too big"
  (x < MAX_STATE_VALUE);
```

Figure 3.5: Murphi code for the FSS in Figure 3.4

```
Startstate startstate fired.
x:0
----------
Rule time step, d:1 fired.
x:1
----------
Rule time step, d:2 fired.
x:3
----------
Rule time step, d:2 fired.
The last state of the trace (in full) is:
x:5
----------
```

Figure 3.6: Murphi error trace for murphi model in Figure 3.5

Cache and Disk Verification. Many systems to verify can result in a (very) big state space (we have just told about state explosion). To this aim, CMurphi is able to use a *cache* or a *disk* memory during the verification to store states. More precisely, in the *Cache Mode*, it uses a cache to store the visited states and can extend the verification queue to disk (the cache *collision rate* can be monitored to stop the exploration if the cache becomes ineffective). Similarly, when the *Disk Mode* is active, the verifier uses the disk to store the visited states and the verification queue instead of a cache memory.

Real Number Support. Murphi built-in types are ranges of integers and enumerative types. To ease the hybrid systems modelling activity we also want to be able to handle *finite precision real numbers* within Murphi, i.e. numbers of the form $s_M d_0.d_1 \cdots d_{m-1} \times 10^{s_E e_{n-1} \cdots e_0}$ where: d_i and e_i are decimal digits, $d_0 \neq 0$, s_M, $s_E \in \{'+', '-'\}$. As usual we call $s_M d_0.d_1 \cdots d_{m-1}$ the *mantissa* and $s_E e_{n-1} \cdots e_0$ the *exponent* of the number $s_M d_0.d_1 \cdots d_{m-1} \times 10^{s_E e_{n-1} \cdots e_0}$.

To this aim, CMurphi allows the use of the type `real(m, n)` for real numbers with *m* digits for the mantissa and *n* digits for the exponent. Type `real(m, n)` is finite, its cardinality is $2 \times 9 \times 10^{m-1} \times 2 \times 10^n = 36 \times 10^{m+n-1}$. This extension has no impact on Murphi verification algorithms (e.g. as that in Algorithm 1), however makes it easier to model hybrid systems.

Note that, as from Remark 3.3.1, the huge cardinality of the type `real(m, n)` does not imply, *a priori*, a huge size of the set of reachable states.

The type `real(m, n)` is built on `long double` C type. For this reason the *mantissa* size *m* and the *exponent* size *n* in `real(m, n)` must satisfy the following constraints: $1 \leq m \leq$ `LDBL_DIG`, $2 \leq n \leq \lfloor \log_{10}$`LDBL_MAX_10_EXP`$\rfloor + 1$, where `LDBL_DIG` is the maximum number of digits for the mantissa of the `long double` C type and `LDBL_MAX_10_EXP` is the maximum value of the exponent of the `long double` C type. These constants are defined in the C header `float.h`. CMurphi also allows to import all functions available in the C `math` library (header `math.h`). Such functions can be freely used within the Murphi input language.

C/C++ External Functions. However, this is not enough to model complex behaviors.

Indeed, the implementation of complex systems requires more advanced language constructs to be described, such as the one provided by the C/C++ language. Moreover, system simulators written in C/C++ are often available (especially in the more complex cases) for testing purposes, thus it is worth doing to reuse them in the verification phase.

To overcome these difficulties, and to reuse simulators, CMurphi allows the use of externally defined C/C++ functions in the modelling language. In this way, one can use the C/C++ language constructs to model complex dynamics. Moreover, one can directly include (with some arrangement) in the Murphi model a simulator for the system under analysis, since a system simulator is almost always written in C/C++.

CHAPTER 4

SYMBOLIC MODEL CHECKING

Symbolic Model Checking is characterised by the application of the model checking technique to a system having states represented in a compressed form (e.g., using BDDs). Moreover, the symbolic approach often allows one to specify an invariant property by using temporal logics (e.g., CTL or LTL), which we shortly introduce in the following.

4.1 CTL AND LTL

Temporal logics are useful to describe properties that hold on an infinite execution path of the system, i.e., the execution path contains a loop. Two kinds of temporal logics are Computation Tree Logic (CTL) and Linear Temporal Logic (LTL), whose union composes the CTL* logic. CTL and LTL differ in how they handle branching in the computation tree. In CTL temporal operators quantify a formula *over the paths* departing from a given state (through universal (**A**) and existential (**E**) quantifiers) . In LTL operators are intended to describe properties of *all* possible computation paths (i.e., all path are universal quantified). The syntax of CTL and LTL formulae obeys to the following rules:

- any atomic proposition is a CTL (LTL) formula;
- if p and q are CTL (LTL) formulae, then $p \cdot q$ and $\neg p$ are CTL (LTL) formulae, where $\cdot$ is any boolean connective (e.g., $\wedge, \vee$).
- if p and q are CTL formulae, then $\mathbf{EX}p, \mathbf{EG}p, \mathbf{E}[p\mathbf{U}q]$ are CTL formulae.
- if p and q are LTL formulae, then $\mathbf{X}p, \mathbf{G}p, [p\mathbf{U}q]$ are LTL formulae.

Intuitively, the meaning of CTL formula $\mathbf{EX}p$ is that there exists $(\mathbf{E})$ a path starting from an initial state in which in the next $(\mathbf{X})$ state p holds. $\mathbf{EG}p$ means that there exists a path starting from an initial state which globally $(\mathbf{G})$ p holds. $\mathbf{E}[p\mathbf{U}q]$ there exists a path starting from an initial state in which p holds until $(\mathbf{U})$ q holds.
All the other CTL operators (e.g., $\mathbf{AF}p$, meaning for all paths eventually hold p) can be derived from the following equivalence rules:

$\mathbf{AX}p \equiv \neg\mathbf{EX}\neg p$	For all paths, in the next state p holds
$\mathbf{EF}p \equiv \mathbf{E}[\top \mathbf{U} p]$	There exists a path in which eventually p holds
$\mathbf{AG}p \equiv \neg\mathbf{EF}\neg p$	p is an invariant
$\mathbf{A}[p\mathbf{U}q] \equiv \neg\mathbf{E}[\neg q\mathbf{U}\neg p \wedge \neg q \wedge \neg\mathbf{EG}\neg q]$	For all path, p until q.

It is worth noting that LTL and CTL have different expressive powers and then they are incomparable (e.g., there is no CTL formula that is equivalent to the LTL formula $\mathbf{AF}(\mathbf{G}p)$). For a survey on LTL and CTL differences the reader can refer to [38].

4.2 Symbolic State Representation

As said in the previous chapter, *state explosion* represents the most ineliminable event that affects all the model checking techniques. Due to this problem, many approaches exploit the use of heuristics as well as new data structures to defer (or avoid in lucky cases) the state explosion.

Symbolic graph algorithms work on an implicit description of the state space, on the contrary of an explicit one, as we have seen in Chapter 3. To give an example, if an integer variable x has 90 different values, a symbolic representation of x may be $0 \leq x \leq 89$ which compactly describe 90 different states through a single integer region. Intuitively, thanks to this representation, all the classical set operators as $\cup, \cap, \subseteq, \in$ can be used to each region of the state space. Hence, modelling the transition function of a system as a *boolean* function, we can represent the state space through a symbolic representation as a DAG (directed acyclic graph). This is the idea behind the symbolic representation of the state space.

4.3 BDDs: Binary Decision Diagrams

A Binary Decision Diagram [29] is a data structure able to represent a boolean function as a DAG. More precisely, each boolean function can be represented as a binary tree having two kind of leaf values: true ($\top$) and false ($\bot$). The terminal nodes are either true or false whilst each non-terminal node (the *decision node*) is associated to a variable of the f formula. Since the graph is binary, each node has exactly two outgoing edges which represent the assignment value for the node variable (i.e., true or false respectively). Intuitively, a path on this graph represents an assignment sequence for the variable of the formula f, as depicted in Figure 4.1.

4.3.1 OBDD: THE BDDS VARIABLES ORDERING

When it is possible to define a total order on the decision node (i.e., all the nodes on the same tree level refer to the same variable) the BDD is called *Orderer* BDD, OBDD hereafter. Usually, OBDDs refer also to *reduced* (ROBDDs) which are obtained by merging isomorphic subgraphs and eliminating all nodes having two isomorphic children, as shown in Figure 4.1. More precisely, the canonicity of BDDs follows by (1) imposing a total order $<$ over the variables set of the decision node (if n has a non-terminal child then $var(n) < var(m)$) and (2) by requiring that the BDD contains no isomorphic subgraphs.

It is clear that the size of a reduced OBDD depends on both the function represented and on the chosen ordering of the variables. Unfortunately, the problem to find the *best* total ordering on variables is NP-hard, even though exist efficient heuristics to handle this problem.

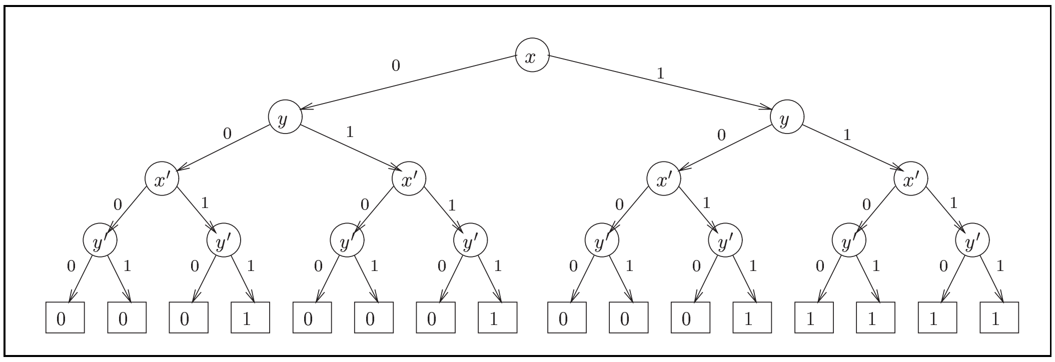


Figure 4.1: Binary Decision Diagram for $f = (x \wedge y) \vee (x' \wedge y')$

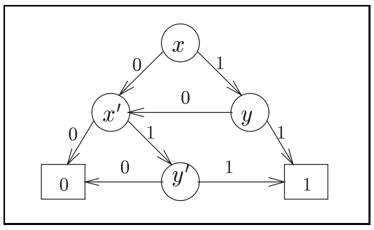


Figure 4.2: Reduced Ordered Binary Decision Diagram for $f = (x \wedge y) \vee (x' \wedge y')$

4.4 THE NUSMV VERIFIER

In this section we briefly introduce the NuSMV [34] tool, a general purpose symbolic model checker based on SMV [154]. Recently, NuSMV2 extended the previous versions of NuSMV with the capability to combine BDD model checking and SAT-based Bounded Model Checking [35]. The NuSMV input language is designed to allow the description of FSSs. The only data types provided by the language are booleans, bounded integer subranges, and symbolic enumerated types, which can be further extended with the definition of bounded arrays of basic data types. The system description is decomposed into *modules*, each of them represent a FSS and can be instantiated many times. Moreover, the NuSMV input language allows one to describe deterministic and non deterministic systems, as well as synchronous and asynchronous systems.

Figure 4.3 we provide an example of NuSMV module. It describes a system with one module (i.e., one FSS), whose state is composed by two variables (*request* and *state*). In the initial state the system is *ready* whilst the value of variable *request* is undefined. The system dynamics is defined by function *next* which describes how the variable *status* evolves. The `case` statement is evaluated from top to bottom. When the *request = true* the system becomes busy, otherwise the system will be non-deterministically *ready* or *busy*. A CTL (and also LTL) formula is specified through the keyword `SPEC`. In this case, the system holds that *"for all paths, at each time step, whenever a request is made then it will be always satisfied in the future"*. A complete tool description can be found at [139].

```
MODULE main
VAR
  request : boolean;
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & (request): busy;
                   1 : {ready,busy};
                 esac;
SPEC
  AG((request) -> AF state = busy)
```

Figure 4.3: Example of NuSMV domain

4.5 THE HYTECH VERIFIER

HYTECH [88] is a symbolic model checker for *linear* hybrid automata, a subclass of hybrid in which the dynamics of continuous variables are defined by linear differential inequalities. One of the most important (and unique) feature of HYTECH is the *parametric analysis*: it uses symbolic constants with unknown fixed values (i.e., design parameters) to determine necessary and sufficient constraints on the parameters under which safety violations cannot occur (e.g., this feature can be used to determine the minimum and maximum bound on variables). Moreover, HYTECH is able to verify system properties expressed by means of LTL logics.
On the negative side, HYTECH can deal only with small automata and no simulation mode is available during the system analysis. For the sake of simplicity, we give provide a brief example of a HYTECH model, a complete tutorial can be found at [89].

Generally, a HyTech model consists of two parts. The former contains the textual description of a *collection* of linear hybrid automata, which are automatically composed for the analysis. The former contains a sequence of analysis commands. The analysis language is a simple while programming language that provides as primitive the data type state assertion with a variety of operations (i.e., *pre* and *post* functions, boolean operator, existential qualification).

Let us to consider a system composed by a train, a gate and a controller. The train is initially some distance away from the track intersection with the gate fully raised (e.g., at least 2000 feet). As the train approaches (1000 feet), it triggers a sensor signaling its upcoming entry to the controller. The controller sends a lower command to the gate, after a delay of up to α seconds. When the gate receives a lower command, it lowers at rate of 9 degrees per second. After the train has exited the intersection and is 100 feet away, it sends an exit signal to the controller. The controller then commands the gate to be raised. HyTech performs symbolic verification based on *regions*. In this case, we can specify:

- The `init` region, defined as `init := loc[train]=far & x=0 & loc[gate]=up & a=9`.
- The `access` region, which describes the reachable states from `init`, as `access := reach forward from init endreach;`.
- The error region to check that the gate is closed when the train is inside the crossing as `Err:= (loc[train]=on) & (~(loc[gate]=down));`

In our example, HyTech requires 7 steps to verify that the specified safety properties is hold for the system.

CHAPTER 5

RELATED AREAS

In the following we briefly introduce three methodologies, namely the *Mixed Integer Non-Linear Programming*, the *Dynamic Programming*, and the *Cell Mapping*, which are applied in many fields to deal with systems having a continuous and nonlinear dynamics.

5.1 MINLP: MIXED INTEGER NONLINEAR PROGRAMMING

Mixed Integer Nonlinear Programming (MINLP) is a mathematical programming which involves continuous and discrete variables and it is characterized by nonlinearities in the objective function and constraints.

MINLP is used in several applications, including the VLSI manufacturing areas, engineering, management science and operations research (a large collection of them can be found in [84] and [85]) since it combines simultaneously the optimization of a *discrete* system structure and *continuous* parameters. The general form of a MINLP is

$$\begin{array}{ll} \min & f(x,y) \\ s.t. & g_j(x,y) \leq 0, j \in J \\ & x \in X, y \in Y \end{array}$$

where $f(x,y)$ is the objective function, $g_j(x,y)$ (for $j \in J$ with J the index set of inequalities) are constraint functions and x and y are the continuous and discrete variables, respectively. The sets X and Y are bounding-box-type restrictions on the variables.

MINLP problem combine two different subproblems: the nonlinear programs (NLP) subproblem and the mixed integer programs (MIP) one. Indeed, MINLP problems are hard to solve since they combine all the difficulties of the combinatorial nature of MIP and the difficulty in solving non-convex (and even convex) NLP. Since subclasses MIP and NLP are among the class of theoretically difficult problems (NP-complete), it follows that solving can be a daring challenge.

There are several methods to solve MINLP problems: the branch and bound method (BB), Generalized Benders Decomposition (GBD), Outer-Approximation (OA), LP/NLP based branch and bound, and Extended Cutting Plane Method (ECP).

To this regard, MINLP has been used in a number of industrial case studies, which typically present a nonlinear dynamics, in both planning and control fields [147, 142, 164, 6].

Although many commercial as well as academic solvers have been implemented (a complete survey can be found in [31]), the application of MINLP to systems having huge state space may be difficult due to the large number of variables and constraints which can compose the MINLP formulation.

5.2 Dynamic Programming

Dynamic programming techniques are very suitable for the generation of (optimal) controllers.

For the sake of brevity, in this section we only briefly recall the main characteristics of the approach, we refer the reader to [20] for a complete description of this widely used technique. Furthermore, in the following we use [114] as a reference point, since it contains a complete theoretical treatment of the problem and also illustrates a detailed algorithm for the numerical synthesis of the corresponding controller.

Consider a nonlinear plant $\mathcal{P}$

$$x_{t+1} = f(x_t, u_t) \tag{5.1}$$

with state $x_t \in \mathbb{R}^n$, control $u_t \in \mathbb{R}^m$ and discrete time $t \in \mathbb{Z}^+_{\not 0}$. It is assumed that $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is continuous.

The problem of the controllability of $\mathcal{P}$ to the *origin* (i.e., the setpoint) is considered in a given bounded region G, containing a neighborhood of the origin itself.

The classical dynamic programming approach proceeds as follows. First an optimal cost function J is considered:

$$J(x) = \inf_{\underline{u}} \left[\sum_{t=0}^{\infty} l\left(F(x, u_t), u_t\right) \right] \tag{5.2}$$

where $l(x, u)$ is a continuous, positive definite *cost function* and $\underline{u}$ stands for a generic control sequence: $\underline{u} = \{u_0, u_1, u_2, \ldots\}$.

J is well defined (i.e. the infimum always exists in the region of interest) if and only if the plant $\mathcal{P}$ is controllable. In this case, J satisfies the so-called *Bellman Equation*:

$$J(x) = \inf_{u} \left[l(x, u) + J\left(F(x, u)\right) \right] \tag{5.3}$$

and it can be computed by the following iterative method:

$$\begin{aligned} & J_0 = 0 \\ & J_{T+1}(x) = \inf_u \left[l(x,u) + J_T \left(F(x,u) \right) \right] \end{aligned} \tag{5.4}$$

where $T \in \mathbb{Z}_0^+$.

Since the convergence of (5.4) poses several problems, in [114] the numerical design of the controller is based on a set of simplifying assumptions.

Indeed, in [114] the authors assume to have a continuous, positive definite *terminal cost function* $\bar{V}(x)$ and that there exists a bounded region $\bar{\Gamma}$, which includes G, such that:

$$\forall x \notin \bar{\Gamma}, u \in \mathbb{R}^m.\ l(x,u) > \bar{V}(x) \tag{5.5}$$

Other conditions are imposed on both $l(x,u)$ and $\bar{V}(x)$, that here are omitted for brevity.

With these assumptions, it is possible to define an *extended cost function* $V(x)$ as follows:

$$V(x) = \inf_{\underline{u}, t' \in \mathbb{Z}_0^+} \left[\sum_{t=0}^{t'-1} l\left(F(x,u_t), u_t \right) + \bar{V}\left(F(x,u_{t'}) \right) \right] \tag{5.6}$$

where $\underline{u}$ stands for a generic control sequence: $\underline{u} = \{u_0, u_1, u_2, \ldots\}$.

Thus, $V(x)$ goes through all possible *finite time horizons* with terminal cost $\bar{V}$ and takes the infimum. The role of functions $l(x,u)$ and $\bar{V}(x)$ in (5.6) can be informally explained as follows: by the assumption (5.5) above, the final cost $\bar{V}$ can be considered as a *penalty function* which *punishes* a wrong control move; it regulates the convergence of the iterative computation of $V(x)$, since such iteration either stops with some final cost or the final cost diminishes further and further, as the minimal cost sequence drives to (a small neighborhood of) the set point.

The main results in [114] are the following:

Proposition 1. *$V(x)$ is continuous and satisfies the following Bellman-like equation:*

$$V(x) = min \left\{ \bar{V}(x), \inf_u \left[l\left(F(x,u_t), u \right) + V\left(F(x,u) \right) \right] \right\} \tag{5.7}$$

Now let $\gamma = \sup_{\xi \in G} V(\xi)$ and $\Gamma' = \{x \in \mathbb{R}^n | V(x) \le \gamma\}$. Γ' is the region where we expect that control trajectories (if they exist) evolve.

Proposition 2. *If for every $x \in \Gamma'$, $V(x) < \bar{V}(x)$ holds, with the exception of a (small) neighborhood $N(\rho)$ of the origin, then in G the plant $\mathcal{P}$ is controllable in the sense that it can be driven to the neighborhood $N(\rho)$ of the origin. Moreover the control sequence can be determined as the minimal cost trajectory, from the equation (5.7).*

From these theoretical results directly derives the following algorithm.

Given the discrete set of points $z_1, z_2, \ldots, z_{\bar{n}}$ contained in $\bar{\Gamma}_D$ and the discrete set of control actions $u_1, u_2, \ldots, u_{\bar{m}}$ contained in $\bar{U}_D$, we have the design algorithm shown in Procedure 2.

Procedure 2 Dynamic Programming Algorithm

```
1: for all $i \in [1, \bar{n}]$ do
2:   $W_0(z_i) \leftarrow \bar{V}(z_i)$;
3:   for all $i \in [1, \bar{n}]$ do
4:     $W_{T+1}(z_i) \leftarrow \min_{j \in [1, \bar{m}]} \{W_T(z_i), \min[l(z_i, u_j) + I(W_T, F(z_i, u_j))]\}$
5:     $t \leftarrow t + 1$;
6:   end for
7: end for
```

Note that, although successfully applied, this technique requires the definition of design functions (like $\bar{V}(x)$ and $l(x,u)$ in Procedure 2) which have to be found out case by case. Moreover, systems whose dynamics cannot be easily inverted (the typical situation for hybrid nonlinear systems) are very difficult to address with the dynamic programming.

5.3 Cell Mapping

Cell mapping was proposed by Hsu [98, 100] as a computational technique for analysing the global behaviour of nonlinear systems.

Cell mapping allows an approximated analysis of a state space by partitioning it into a finite number of disjoint cells. Thus, each variable ranges on the set of cells, instead of $\mathbb{R}^n$. More precisely, suppose to have a model of the form $x(t+1) = f(x(t))$, where the state x is described by n real-valued variables. Then, we can see x as a point of $\mathbb{R}^n$. In the cell mapping, the n axes of the state space are partitioned into equal intervals, each denoted by an integer z_i.

These axes partitions naturally define n-dimensional *cells*. Indeed, a cell z is defined as a n-tuple of intervals $z = [z_1, \ldots, z_n]$. The union of all cells z is the cell space Z. The main effect of cell partition is that all elements in a cell z_i are approximated with the cell center point z_i^c. This allows real (or point to point) trajectories in the state space to be approximated by cell trajectories in the corresponding cell space. Figure 5.1 illustrates the approximation scheme for a real trajectory:

$$x_1 \rightarrow x_2 \rightarrow x_3$$

for the discrete time system:

$$x_{k+1} = f(x_k).$$

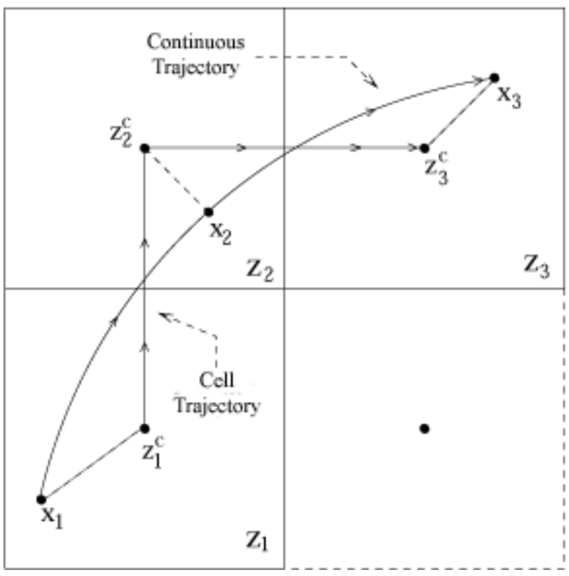


Figure 5.1: Cell space approximation of a real trajectory

The initial point x_1 in the trajectory which lies in cell z_1 is abstracted by the cell center point z_1^c. Then, $x_2' = f(z_1^c)$ which lies in cell z_2 is abstracted by z_2^c. Finally, $x_3' = f(z_2^c)$ in cell z_3 is abstracted by z_3^c. This procedure yields the cell trajectory:

$$z_1 \to z_2 \to z_3$$

Note that to minimise cell mapping errors, it is important that states x_2' and x_3' be located as close as possible and lie in the same cells as the real trajectory states x_2 and x_3, respectively.

A cell mapping is formalized as a cell state space function:

$$C : Z \to Z$$

Using this function, a k-step trajectory emanating from cell z is written as a cell sequence:

$$z \to C(z) \to C(C(z)) \to \ldots \to C^k(z).$$

A periodic motion with period K is a sequence of K distinct cells $z_m, m = 0, \ldots, K-1$, satisfying the condition

$$z_m = C^m(z) \text{ and } z = C^K(z).$$

An *equilibrium cell* z_e is a cell that maps to itself, i.e.:

$$z_e = C(z_e).$$

It is a periodic motion with period 1. The *r-step domain of attraction* of a periodic motion is the set of all cells that are within r-steps of the periodic motion.

The cell map of a system is constructed using an *unravelling algorithm* to compute cell trajectories [100]. Based on these trajectories, one can establish which cells converge to the setpoint (controllable cells) and which not (uncontrollable cells). Moreover, cell mapping has been used to generate optimal control table directly as a controller (see, e.g. [99]), or to fine-tune a fuzzy logic controller (see, e.g. [102]). Furthermore, cell mapping can be used to evaluate controller performance [141, 140].

For the sake of brevity, we refer the reader to [98] for further details about cell mapping and applications. Here we only highlight that cell mapping requires a *global* analysis of the state space, and thus, for complex systems, when a high precision is required, cell mapping is *hard to apply*.

Part II

Explicit Model Checking for the Analysis of Deterministic Systems

In the second part of the Thesis, we show how the explicit model checking technique can be used to analyse *deterministic systems*. In particular, in Chapter 6 we first introduce the problem of Planning and Universal Planning providing a survey on the state of the art. Then we present the contributions that this thesis gives to UPMurphi and, in general, to the planning and control communities.

In Chapter 7 we describe the improvements of UPMurphi: the V-UPMurphi tool which exploits the disk based technique to perform Universal Planning on systems having a big reachable state space. In Chapter 8 the application of V-UPMurphi to some real-world problems is detailed. Finally, in Chapter 9 we show how the explicit model checking technique can be used to perform *data quality analysis* on dirty database, providing a methodology and showing first experimental results on a real case scenario.

CHAPTER 6

MODEL CHECKING BASED CONTROL OF DETERMINISTIC SYSTEMS

6.1 INTRODUCTION

In following we first introduce the problem of Planning and Universal Planning and we illustrate the state of the art. Then, we formally define the universal planning problem on FSS (according to the Definition 2), and we describe the algorithm as presented in [49, 56] that solves this problem by means of a model checking derived algorithm.

Planning and Scheduling. For many years, planning and scheduling research were completely separated. Typically *planning* concerns the problem of generating a sequence of actions, in order to move from a specified initial state to a desired goal state. Differently, *scheduling* is interested in allocating known activities to available resources and time respecting capacity, precedence and other constraints, optionally minimising a given cost function. In pure-scheduling problems often there are many ways to accomplish the same task, synthesising a schedule "as long as possible". To give an example, the Job Shop problem is a typical scheduling problem where the goal is to *allocate* jobs to machines, minimising a cost function (e.g., the workload of the machines). The scheduler will try many possible allocations looking for the best one.
Differently, in pure-planning problems the system is described through a *dynamics* that should be discovered. The *blocks world* example is a pure-planning problem which describes the world as composed by blocks and a robot arm. The arm can perform some actions on blocks and the goal is to create an ordered stack of blocks.

Generally, a planning problem requires (1) an initial state, (2) a description of the system dynamics, (3) a description of the goal and the result is a sequence of actions (i.e. the *plan)*. A scheduling problem requires (1) a set of activities with preferences, (2) a set of available resources and the result is a map between activities and resources. In spite of the differences that branch planning and scheduling, recently a mutual interest between planning and scheduling has emerged since planning algorithms have been applied to many real-world problems. To give an example, many planning domains require to deal with limited resources. As a consequence, many communities have merged in a common research (see, e.g., the International Conference on Automated Planning and Scheduling [1]).

Universal Planning. If by one hand planning concerns the run-time generation of plans to be applied between a single source and a single goal, on the other hand a *universal plan* can be seen as a collection of plans (or a set of policies) able to bring the system to the goal from any feasible state. The concept of *Universal Planning* was first introduced by Schoppers [150] as an approach to learn state-action rules in which a plan represents a solution path for all possible configurations of a planning problem, instead of a solution for one single initial state. The solution of the universal planning problem is a Universal Plan, which summarises the commands (actions) to send to the plant in order to reach a goal from any possible state the plant can be in.

It is worth noting that universal planning is typically performed off-line and is computationally much harder than planning. However, once a universal plan is computed, its reaction time will be very small when compared to that of planning.

This Chapter is devoted to the automatic generation of *optimal* universal plans through explicit model checking.

The main idea of Schoppers to synthesise a universal plan is to perform a backward visit of the dynamics graph (starting from the goal nodes) using a classical BF search. However, for many years the problem to find a universal plan for a given domains, even though it is considered interesting for the large application in many problems, it has been deemed impracticable [81] since the search tree can grow exponentially in the size of the graph.

6.1.1 Related Work

With the introduction of BDD state representation (see Section 4.2 for details) the concept of universal planning has a renaissance in both deterministic and non-deterministic domains, thus many planners and universal planners based on model checking (and hence on the formalism provided by FSS) have been proposed. To this regard, planning-as-model-checking has a strong heritage (see, e.g., [83]), since proving states reachability can be viewed as finding plans. In particular, in [37] the authors use a symbolic approach based on OBDDs to compact encode the state space.

Dplan [149] is well known state-based, *backward* universal planner for deterministic domains. The main characteristics are that (1) it represents a state explicitly, (2) no initial states are given and (3) it performs a backward search starting from the goals nodes. The construction of the universal plan terminates only if the expansion of the current leaf node results in a yet visited state. However, due to the backward search, an inverse operator op^{-1} should be defined for each domain operator op. Thus, Dplan does not work well on systems whose dynamics is difficult to invert (the typical situation for hybrid nonlinear systems).

The Model Checking Integrated System (MIPS) [62, 63] is a very powerful and complex framework that makes use of a combination of both explicit and symbolic model checking based on heuristic search. The MIPS performed very well in different planning competitions, however it is restricted to PDDL2.1 while the extended version MIPS-XXL [66] deals with PDDL3. However, MIPS it is not a universal planner.

The UPPAAL/TIGA tool [15] is built on top of UPPAAL which allows the use of real variables only as clocks, thus excluding systems with nonlinear dynamics. Other examples of model checking based deterministic planners include, among others, ProPlan [73] and BDDPlan [92].

In [36] authors use a symbolic (OBDD-based) model checking approach to synthesise optimal (with respect to the length of the plan) universal plans for non-deterministic plants (we will detail this work in the Part II of this thesis).

However, all these approaches require the explicit definition of an inverse function for each operator used in the domain, and thus their application is hard when dealing with systems having a complex and nonlinear dynamics.

Indeed, a growing number of motivating applications shows the importance of dealing with mixed discrete continuous domains. Some examples are: product processing in a plant [9], activity management of an autonomous vehicle [119], voltage regulation planning [16], solar array operations on the International Space Station [145], oil refinery operations planning [23], planning for an airport control system [90], or slag foaming control [168].

To this regard, several real world planning problems present complex *nonlinear* behaviours which are difficult to handle by any analytical method (see, e.g., [155], [22], [26]) or hybrid reasoning approach. Nonlinearity can arise from the intrinsic dynamics of the system (e.g., the regulation of a steering antenna, which leads to an inverted pendulum problem), or the saturation of actuators (e.g., valves that cannot open more than a certain limit, control surfaces in an aircraft that cannot be deflected more than a certain angle, etc.). Indeed, the behaviour of nonlinear systems can be so complex to be completely unpredictable after a small interval of time (see, e.g.,[151]).

To this aim, the planning community has made a great effort to develop algorithms and tools able to deal with hybrid planning domains, which can be modelled via the PDDL+ language, as discussed in Section 2.4.

Continuous Linear Domains. In such a context, it is crucial to reason about continuous change during the planning process [75]. In addition to the model checking based planners cited above, other planners able to deal with hybrid domains have been proposed. More recent works include the OPTOP planner [128] that deals with linear continuous domains where concurrent processes do not affect the same variable.

The TM-LPSAT system, developed by [152], combines SAT and LP solvers. The former is used to deal with the discrete component of the domain while the latter is used to handle the continuous one. TM-LPSAT can deal with processes modelled in PDDL2.1, even though it is limited to small linear problems.

COLIN [42] is a powerful tool for planning in domains with linear continuous processes. It extends the forward chaining temporal planner CRIKEY3 [41], making it able to reason with actions with continuous linear effects. COLIN integrates a guided state space search with linear programming, and supports duration-dependent effects, durative actions with continuous change and concurrent continuous change.

However, the planners above are not universal planners and they can handle only linear domains.

Continuous Nonlinear Domains. Looking at planners able to deal with nonlinear dynamics, we highlight Kongming [119, 122], thanks to the concept of Flow Tubes, is able to compactly represent hybrid plans and encode hybrid flow graphs as a mixed logic linear/nonlinear program, solvable using an off-the-shelf solver. However, Kongming can only address planning problems with constant action duration (e.g., consider a pump that fills a tank, the duration of the ”fill” action cannot depends on the tank’s volume).

More recently, [134] deals with nonlinear continuous effects written in PDDL+, using a state projection algorithm implemented into a Hierarchical Task Network planner. The approach is very interesting and effective, even though no information about the optimality of the synthesised solutions is given in the paper and, as the authors argue, the scalability of their approach has not yet been evaluated on more complex case studies.

Thus, planning as well as universal planning with continuous nonlinear change is a challenging issue.

6.2 THE UNIVERSAL PLANNING PROBLEM

In order to formally define the universal planning problem for continuous systems with possibly nonlinear dynamics, we assume that a set of *goal states* $G \subseteq S$ has been specified. Moreover, to have a finite state system, we fix a *finite temporal horizon* T and we require each plan to reach the goal in at most T actions. Note that, in most practical applications, we always have a maximum time allowed to complete the execution of a plan, thus this restriction, although theoretically quite relevant, has a limited practical impact.

For the sake of completeness, in the following we provide the formal definitions of Planning Problem on FSS and its solution.

Definition 11 (Planning Problem on FSS)**.** *Let* $\mathcal{S} = (S, s_0, A, F)$ *be an FSS. Then, a* planning problem *(*PP *in the following) is a triple* $PP = (\mathcal{S}, G, T)$ *where* $s_0 \in S$, $G \subseteq S$ *is the* *set of the goal states, and T is the finite temporal horizon.*

Then, a solution for PP is a reachable trajectory π *(*plan*), according to Definition 4, $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \ldots a_{n-1} s_n$ where, $\forall i = 0, \ldots, n-1$, $s_i \in Reach(S)$ is a state, $a_i \in A$ is an action, $F(s_i, a_i, s_{i+1}) = 1$, $|\pi| \leq T$, and $s_n \in G \subseteq Reach(S)$.*

Now we are in position to state the universal planning problem for FSSs.

Definition 12 (Universal Planning Problem on FSS)**.** *Let $\mathcal{S} = (S, I, A, F)$ be an FSS. Then, a* universal planning problem *(*UPP *in the following) is a quadruple $\mathcal{P} = (\mathcal{S}, G, C, T)$ where $G \subseteq S$ is the set of the goal states, $C : S \times A \to \mathbb{R}^+$ is the cost function and T is the finite temporal horizon.*

Intuitively, a solution to an UPP can be seen as a set of *policies*, that is a set of minimal cost paths in the system transition graph, starting from any reachable system state and ending in a goal state.

More formally, we have the following:

Definition 13 (Solution for UPPs)**.** *Let $\mathcal{S} = (S, I, A, F)$ be an FSS and let $\mathcal{P} = (\mathcal{S}, G, C, T)$ be an UPP. Moreover, let $\Omega = \bigcup_{s_I \in I} \text{Reach}(s_I) \cap \bigcup_{s_G \in G} \text{Reach}^{-1}(s_G)$. Then a solution for $\mathcal{P}$ is a map $\mathcal{K}$ from Ω to A s.t. $\forall s \in \Omega$ there exist $k \leq T$ and a trajectory π^* in $\mathcal{S}$ s.t.: $\pi^*_s(0) = s$, $\forall t < k : \pi^*_s(t+1) = F(\pi^*_s(t), \mathcal{K}(\pi^*_s(t)))$ and $\pi^*_s(k) \in G$. We denote with $\mathcal{K}_\pi(s)$ the trajectory π^* generated by $\mathcal{K}$ and s.t. $\pi^*_s(0) = s$.*

An optimal solution *is a solution $\mathcal{K}$ s.t. for all other solutions $\mathcal{K}'$ the following holds: for all $s \in S$ s.t. $\mathcal{K}_\pi(s)$ and $\mathcal{K}'_\pi(s)$ are defined, then $C(\mathcal{K}_\pi(s)) \leq C(\mathcal{K}'_\pi(s))$.*

In the next section, we describe the algorithm presented in [49] which takes as input an UPP and outputs an optimal solution for it.

6.3 Planning as Model Checking

In Section 3 we gave the main idea of how a model checker works, (i.e., it looks for a system state which violate an invariant condition). Similarly, a planner performs a search into the system by looking for a state which satisfy a goal condition. Then, both model checker and planner return a path (i.e., an *error trace* and a *plan* respectively) from the initial state to the found state.

It is possible to use a model checker like a planner by forcing the former to look for an invariant which models a goal condition. More formally.

Definition 14 (Planning as Model Checking)**.** *Let MCP be a Model Checking Problem $\mathcal{M} = (\mathcal{S}, \varphi, T)$ according to Definition 10 and let PP be a Planning Problem $P = (\mathcal{S}, G, T)$*

as in Definition 11, we construct a planning as model checking *problem as* $PM = (\mathcal{S}, \psi, T)$ *where* $\psi = s \notin G$.

Then, a solution for PM is a reachable trajectory π *(*plan*), according to Definition 4,* $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \dots a_{n-1} s_n$ *where:* $\forall i \in [0, n-1]$, $s_i \in Reach(S)$ *and* $s_i \models \psi($ *that is* $s_i \notin G)$ *whilst* $s_n \in Reach(S)$ *does not satisfy* ψ *(that is* $s_n \in G$*). If* $\forall s \in Reach(S), s \models \psi$ *then the solution is an empty trajectory (a plan does not exist).*

6.4 THE MODEL CHECKING BASED UNIVERSAL PLANNING ALGORITHM

In this section we describe an explicit model checking based algorithm to perform Universal Planning on continuous domains, as proposed in [49, 56]. Given a UPP, the algorithm solves it in two phases: the BUILDGRAPH and the UPLANGENERATION.

Procedure 3 BUILDGRAPH(UPP $\mathcal{P} = (\mathcal{S}, G, C, T)$)

```
let $\mathcal{S} \leftarrow (S, I, A, F)$
for all $s \in I$ do
  Enqueue($Q_S$, $s$)
  Insert(HT, $s$)
  if ($s \in G$) then
    Enqueue($Q_G$, $s$)
    HT[$s$].cost $\leftarrow 0$
  end if
  while (( $Q_S \neq \emptyset$) $\wedge$ (current_BFS_level $\leq$ T)) do
    $s \leftarrow$ Dequeue($Q_S$)
    for all $s' \in \{F(s,a) \mid (a) \in A\}$ do
      if ($s' \notin$ HT) then
        Insert(HT, $s'$)
        if ($s' \in G$) then
          Enqueue($Q_G$, $s'$)
          HT[$s'$].cost $\leftarrow 0$
        else
          Enqueue($Q_S$, $s'$)
        end if
      end if
      PT[$s'$] $\leftarrow$ PT[$s'$] $\cup \{s\}$;
    end for
  end while
end for
```

6.4.1 The BuildGraph Procedure

In the first phase, the algorithm exploits *reachability analysis* in order to build a representation of the system dynamics that can be later easily analysed during the universal plan generation. Indeed, the corresponding BuildGraph procedure, whose pseudocode is given in Algorithm 3, can be seen as an extension of the common breadth-first visit performed by classical explicit model checking algorithms (see the BF Algorithm 1, given in page 23).

For the sake of completeness, it is worth noting that, in the general theory of universal planning, the concept of start state is not present [150], as discussed in Section 6.1. However, in the practice, the concept of reachable state implies such a start state. In other words, we need to start-up the universal planning with a set of start states, that we call a *start state cloud*. These states should be distributed in the system state space so that all the interesting states are reachable from at least one of them. However, a start state cloud can be also suitably prepared to concentrate the planning process on the most interesting state space regions, or to exclude hardly reachable states from the universal plan. Indeed, a complete universal plan could generally contain many rarely-used plans, whose computation requires however time and space. Therefore, an appropriate formulation of the start state cloud may help to minimise the universal plan generation effort and maximise its usefulness. The role of the start state cloud will result more clear in Chapter 8, when we apply the UPMurphi to real case studies.

The Procedure 3 uses the hash table HT to store already visited states, while the queues Q_S and Q_G store the states to be expanded and the reached goal states (to be used in the next phase), respectively. This information is also used to detect and exploit trajectories intersections, so avoiding work duplication. Note that the computation of the successor states involves discretised values, i.e., continuous components of both s and s' in line 11 of Algorithm 3 are rounded according to the chosen discretisation. Finally, the predecessor table PT contains the immediate predecessors of each visited state. This structure is at the heart of the second phase of the algorithm, represented by the UplanGeneration procedure, whose pseudocode is given in Algorithm 4.

6.4.2 The UplanGeneration Procedure

The UplanGeneration procedure performs another Breadth First visit, this time on the *inverted* transition graph, starting from the reached goal states. To this end, the procedure uses the information in Q_G, HT and PT prepared by BuildGraph. The output is the table UPLAN, containing (state,action) pairs that represent the map $\mathcal{K}$ described in Definition 13.

In particular, the check on line 10 of Algorithm 4, which updates the action associated to a state only if either no action has been defined yet or the current action leads to a better result, together with the ordered insertion in the queue Q_S, guarantee that the algorithm

Procedure 4 UPLANGENERATION

```
1: UPLAN $\leftarrow \emptyset$
2: $Q_S \leftarrow Q_G$ // this erases the previous content of $Q$
3: while $Q_S \neq \emptyset$ do
4:   $s \leftarrow$Dequeue($Q_S$)
5:   prev_cost $\leftarrow$ HT[$s$].cost // 0 if $s \in G$
6:   for all ($\vec{s} \in$ PT[$s$]) // $\vec{s}$ is a predecessor of $s$ do
7:     local_cost $\leftarrow \min_{(a)\in A \mid F(\vec{s},a)=(s)} C(\vec{s},a)$
8:     U $\leftarrow \{a \in A \mid F(\vec{s},a) = s \wedge C(\vec{s},a) = \text{local\_cost}\}$
9:     local_action $\leftarrow$ pick an action in U
10:    if (UPLAN[$\vec{s}$]$= \emptyset \vee$HT[$\vec{s}$].cost $>$ prev_cost$+$local_cost) then
11:      UPLAN[$\vec{s}$] $\leftarrow$ local_action
12:      HT[$\vec{s}$].cost $\leftarrow$ prev_cost + local_cost
13:      Enqueue_in_Order($Q_S$, $\vec{s}$)
14:    end if
15:  end for
16: end while
17: return UPLAN
```

returns an *optimal solution* according to Definition 13.

Note that, since our approach rebuilds the system transition graph by a forward analysis of its dynamics, the system fed to the planning algorithm can be of any complexity, and in particular its transition function can be also very difficult to invert.

6.5 The UPMurphi Universal Planner

For the sake of brevity, in this section we briefly describe the main characteristics of the UPMurphi planner. A complete description can be found at [56]. The UPMurphi tool [49, 56] exploits explicit model checking algorithms (as discussed in Chapter 3) to generate optimal controllers.

The UPMurphi tool is built on top of the CMurphi [32] model checker. A graphical representation of the overall structure of UPMurphi, together with its inputs and outputs, is given in Figure 6.1. Each module of UPMurphi has the following role:

PDDL+ domain and problem. It represent the planning problem encoded in PDDL+ language, as described in Section 2.4.

UPMurphi definition. Alternatively, UPMurphi can works directly on the FSS of the domain using the CMurphi language, as described in Section 3.4.

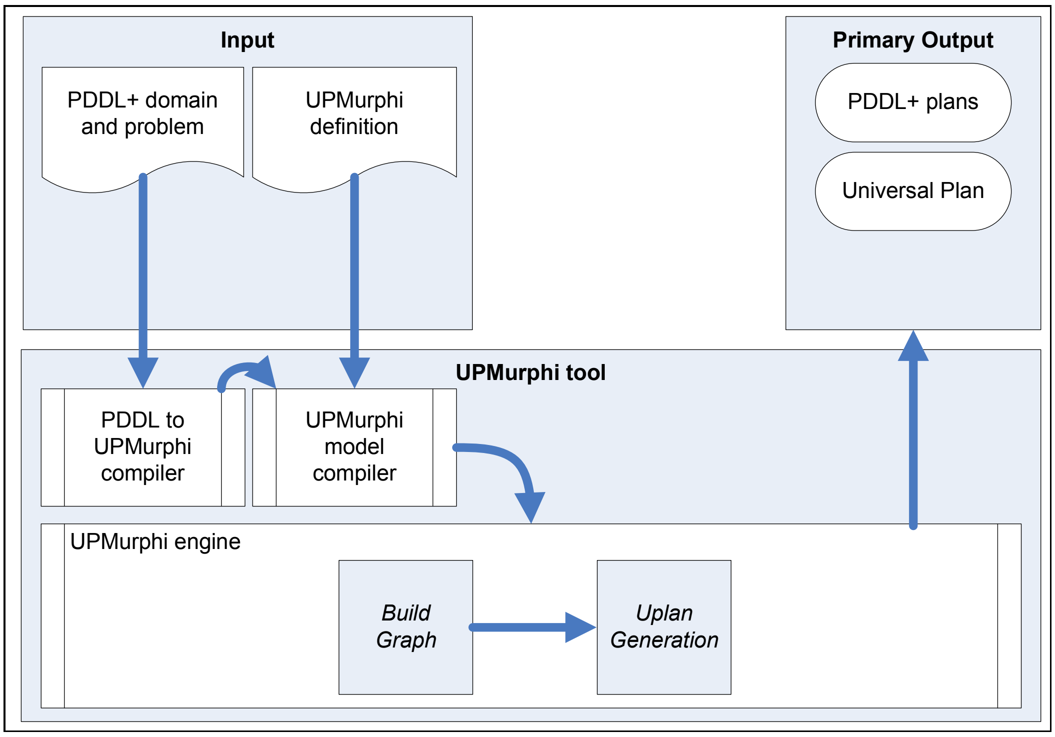


Figure 6.1: Overall structure of the UPMurphi tool

PDDL+ to UPMurphi compiler. This module discretises PDDL+ domains and problems into FSS according to the formal mapping between PDDL+ and FSS semantics, as we provided in [56, 49].

UPMurphi engine. It is the core of the tool and implements the BUILDGRAPH and UPLANGENERATION algorithms on the top of the CMurphi algorithms and data structures. Thanks to this, the universal planning algorithm presented in Section 6.4 can exploit all the CMurphi built-in state space optimisation techniques (such as bit compression [137], symmetry reduction [105], secondary memory storage and state space caching [54]) to handle large systems with huge state spaces.

UPMurphi model compiler. This module compiles the UPMurphi model into an executable model.

It is worth noting that UPMurphi inherits from CMurphi two important features to ease the modelling activity: the type `real(m,n)` of real numbers (with m digits for the mantissa and n digits for the exponent), and the use of externally defined C/C++ functions in the modelling language. In this way, for example, one can use the C/C++ language constructs and library functions to model complex dynamics.

Moreover, to perform the Universal Plan Algorithm as well as to better integrate the Murphi language with PDDL+ input/output syntax, the Murphi input language (described in Section 3.3.1) has been extended as follows:

PDDL name: The modeller can associate to each state variable a PDDL name through the keyword `pddlname`, which will be used by the *UPMurphi model compiler* during the *PDDL+ plans generation* phase. To give an example, the declaration of the state variable `d : 0..1;` becomes `d [pddlname: 'daytime';] : 0..1;`. Moreover, `pddlname` can be used also to specify the PDDL name of a `rule`, which models PDDL+ actions/durative actions and events.

Rules Type: The PDDL+, in contrast with PDDL2.1, models durative actions according to the *start-process-stop* model (introduced in [77], it divides each durative action as start/stop actions, a process which affects continuous variables and an events which models failure conditions). To this regard, the Murphi `rule` construct can be parametrised by specifying one of the following attributes `clock,action,event,durative-start,durative-end`.

Costs and Duration: In order to make UPMurphi able to synthesise *optimal* solution (as described in Section 6.4), the modeller can use the keyword `weight` to specify the *cost* of the action execution as well as the keyword `duration` to (optionally) specify the *duration* of the action.

Metrics: The keyword `metric: {minimize | maximize}` is used to maximize/minimize the cost of the generated solution.

Finally, for the sake of completeness, in Figure 6.2 we figure out the launch options helper of UPMurphi. A detailed description of the formal mapping between Murphi language and PDDL+ has been published in [56].

```
General:
        -search:o     create an optimal plan for each startstate (default).
        -search:u     create an universal plan.
        -search:uo    create an universal optimal plan.
        -search:f     create a feasible plan for each startstate.
Exploration Strategy: (default: -v)
        -cdl          check for deadlock.
        -l<n>         maximum bfs level (default: unlimited).
Output:
        -output file     write output in file (default: stdout).
        -format:pddl     output plans in pddl format (default).
        -format:pddlv    output plans in pddl format with verbose comments.
        -format:pddlvv   output plans in pddl format with very verbose comments.
        -format:text     output plans/actions in text format.
        -format:verbose  output plans/actions in verbose text format.
        -format:raw      output actions in binary format.
```

Figure 6.2: UPMurphi's helper

CHAPTER 7

V-UPMURPHI: THE DISK-BASED ALGORITHM

7.1 CONTRIBUTION

In Chapter 6 we formalised the Universal Planning problem via Finite State Systems, and we also described the algorithm which UPMurphi implements to solve it. In this Chapter, we describe an UPMurphi's enhancement (i.e., V-UPMurphi), which exploits disk storage to extend the applicability of the tool to complex systems. In particular, the disk based algorithm provides the following main contributions:

1. It allows the use of the disk during the exploration of the dynamics. It is worth noting that in Section 3.4 we discussed the CMurphi model checker, which exploits the use of the disk during the verification process (i.e., the *queue* is stored on disk during the verification process). V-UPMurphi goes in the same direction of CMurphi by stressing the use of the disk also during the BUILDGRAPH and UPLANGENERATION phases. Thanks to this approach we are able to synthesise plans and strong plans for some real-world problems, as we discuss in Section 8.

2. It allows one to *pause* the synthesis process. Indeed, it is possible to store the expanded graph to disk and resume the analysis later, also on another machine.

3. It implements an adaptation of the *hash compaction* technique [158] which is compatible with the disk-based algorithm.

4. It now uses the CUDD library [44] to apply OBDD compression on the generated controller. Indeed, the compression technique proved to be very effective for this purpose (see [46] for details).

7.2 THE DISK ANALYSIS ALGORITHM

Figure 7.1 shows the new overall structure of V-UPMurphi in which the *UPMurphi engine* is strictly different with respect to the one presented in Section 6.5 (see Figure 6.1).

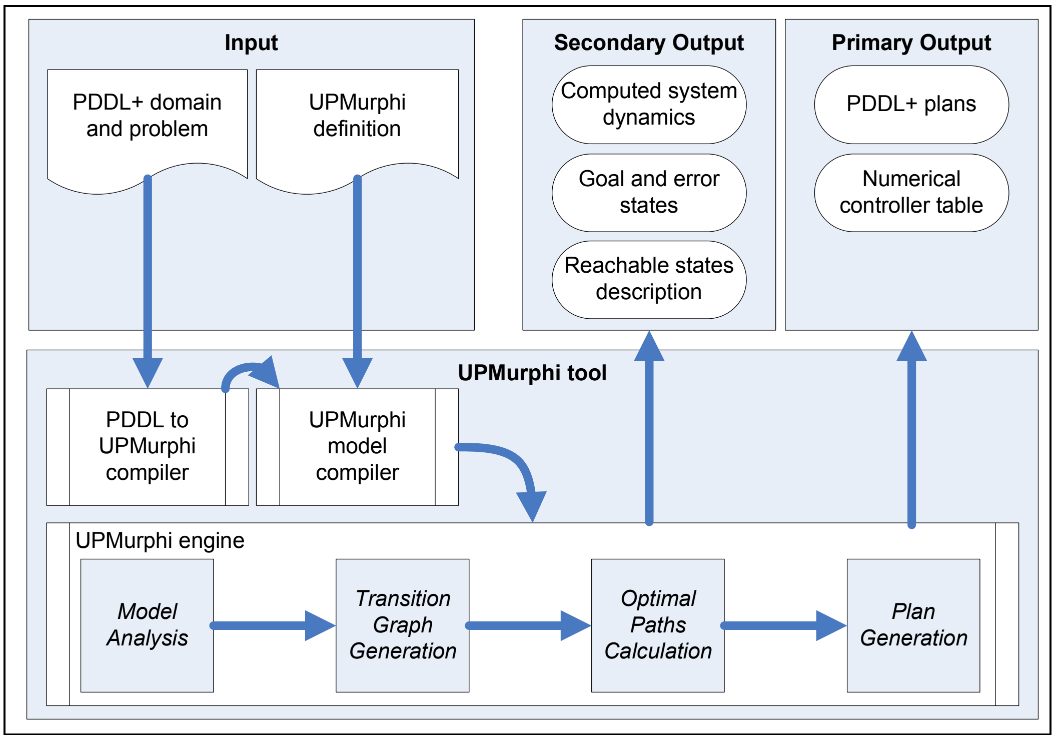


Figure 7.1: Overall structure of the V-UPMurphi tool

Given the model of a system to analyse, the V-UPMurphi engine applies to it an explicit algorithm, organised in four phases, as shown in Figure 7.1.

In the first phase, we exploit *reachability analysis* in order to build a representation of the system dynamics that can be later easily analysed and exploited during the other phases. This phase can be seen as an extension of the common breadth-first visit performed by classical explicit model checking algorithms. In the next phase the tool rebuilds the complete plant transition graph, which is then used to calculate the optimal control paths. Finally, the control paths are grouped into distinct plans, if required. The results can be exported to disk in various formats (binary, PDDL, CSV, etc.).

All the phases above make use of secondary memory (disk) to allow the manipulation of huge systems without incurring in out of memory errors. To avoid an excessive time overhead, the disk structures used by UPMurphi have been designed and implemented by taking into account their usage patterns, i.e., how and how frequently each structure is accessed during each phase of the planning process. This led to the definition of algorithms and data structures that minimise the number of disk seek-and-read operations, which are the bottleneck of any disk algorithm, since seeks suffer from a latency time that is much higher than the actual read/write time. For instance, UPMurphi privileges sequential read/writes, at the cost of duplicating some information and/or requiring more disk space, which is not a problem since large disks are nowadays very common. Moreover, UPMurphi is able to adapt its algorithm to increase or decrease the disk usage with respect to the user specified options and the size of the system under analysis. Finally, when memory storage is absolutely required, the data stored in RAM is compressed. For example, states stored in the memory hash table are written as 40-bit signatures.

Thanks to this framework, the UPMurphi computation can be stopped after each phase and restarted later from the same point, without having to repeat the whole process. Indeed, the data stored in the files above are enough to rebuild the tool state and continue its work. It is even possible to restart the process from a previous phase to try different user settings: for example, one may restart the process from the optimal paths calculation phase (thus reusing the results of the previous two phases), specifying a different transition selection policy. To this regard, the user can specify the `-phase<1..5>` option selecting the appropriate phase to resume (from *model analysis* to *output results* phases).

7.2.1 Disk Data Structures.

The UPMurphi algorithm uses a set of memory and disk data structures. In particular, the only structure that is always stored in memory is the hash table H, used to remember visited states by storing their 40-bit signature and the associated index. The disk structures are described in the following.

Disk Queues. Two FIFO queues Q and Q', used in several parts of the algorithm, are stored to disk. The head and tail segments of each queue are cached in memory to minimise disk accesses. To this aim, the Disk Mode of CMurphi (as described in Section 3.4) has been used and adapted to work on V-UPMurphi.

Reachables file RF. It stores the complete definition of each reachable system state, and it is indexed by the hash table H. This file is created only if the user requires a complete symbolic dump of the state in the planner output.

Transitions file TF. It compactly stores all the transitions encountered during the model analysis. Each entry in this file has the form $(s, d, (r_i, w_i, s'_i)_{i=1\ldots d})$ where s is the index (as stored in H) of a state, d is its out degree, r_i is the index of the model rule which determines the i-th outgoing transition from s, w_i its weight and s'_i the corresponding target state. Note that states are stored as integer indexes, which are usually much smaller than the state description.

Startstates, Goals and Errors files. The *startstates file SF*, *goals file GF* and *errors file EF* contain, respectively, the indexes of the start states and of the reached goal states, and the transitions (described as in the transitions file) which lead to an error.

Actions file AF. The *actions file AF* stores (s, r) pairs where s is a state index and r is the action chosen for that state by the plan generation algorithm. Note that this is an internal encoding of a controller table.

Plans file PF. The *plans file PF* contains strings of the form $s_0 r_0 s_1 r_1 \ldots s_n$, which represent a computed plan from the state (with index) s_0 to state s_n through actions (rules) $r_0 \ldots r_{n-1}$.

Graph file TGF. The *graph file TGF* is used to store the transition graph of the system in the form of adjacency lists. In practice, this file contains an ordered and further compacted representation of the data in the transitions file, to allow a faster navigation of the system dynamics. In particular, for each state (in index order), the file contains an adjacency list composed by (r, w, s') triples, where r is a rule index, w its weight and s' the reached state. The graph file is indexed by an in-memory structure that contains the disk position of the beginning of each list, to allow direct jumps to the adjacency list of a given state.

The transition graph storage *dynamically adapts* to the system size. Indeed, if the number of reachable system states is small enough to allow building the transition graph directly in memory, the graph file is not created and a corresponding memory structure is built instead, to allow faster graph navigation.

In the following we give details of the four algorithm phases, which exploit the disk data structures above.

7.2.2 Model Analysis.

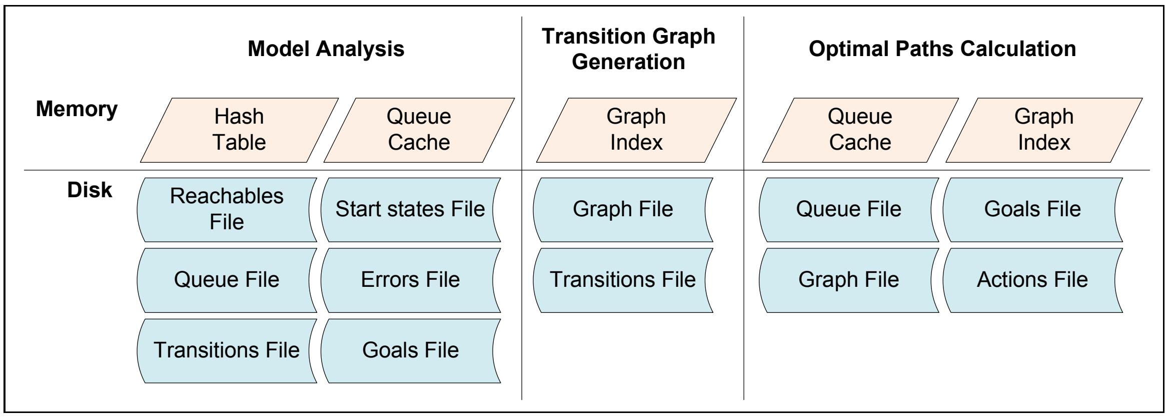


Figure 7.2: Data structures used in the main disk analysis algorithm phases

The model analysis algorithm, i.e., Algorithm 5, exploits a standard BFS search to explore the reachable system states, starting from the given start states. Clearly, when the search reaches an error state or a goal state, it does not explore further on that direction.

The first column of Figure 7.2 shows the memory and disk data structures used by this phase. The disk queue Q is initially loaded with all the start states, whose assigned indexes are also written to the start states file SF. Then the algorithm dequeues a state from Q and visits all its successors, calculated by the `next` function. The exploration ends when there are no further states to expand in the exploration queue.

The memory hash table H is used to avoid revisiting states. Indeed, each time a state is reached by the search procedure, its signature is looked up in H and, if it is not found

Procedure 5 MODEL ANALYSIS

```
Input: $S$, the set of Startstates
$Q \leftarrow \emptyset$;
$H \leftarrow \emptyset$;
for all $s \in S$ do
  $i$ ←generate_index($s$);
  store($H$,signature($s$));
  enqueue($Q, s$);
  write_state_index($SF, i$);
end for
while $Q \neq \emptyset$ do
  $s \leftarrow$ dequeue($Q$);
  outgoing $\leftarrow \emptyset$;
  for all $(r, w, s') \in next(s)$ do
    if not contains($S$,signature($s'$)) then
      $i$ ←generate_index($s'$);
      store($S$,signature($s'$));
      outgoing ←outgoing$\cup(r, w, s')$;
      if is_goal($s'$) then
        write_state_index($GF, i$);
      else if is_error($s'$) then
        write_transition($EF, s, r, w, s'$);
      else
        enqueue($Q, s'$);
      end if
    end if
  end for
  write_state_index($TF, s$);
  write_number($TF, |outgoing|$);
  for all $(r, w, s') \in$transitions do
    write_transition($TF, r, w, s'$);
  end for
end while
```

Procedure 6 TRANSITION GRAPH GENERATION

```
while $(TF$ is not completely read$)$ do
  $s$ ←read_state_index($TF$);
  $n$ ←read_number($TF$);
  for $i = 1$ to $n$ do
    $(r, w, s') \leftarrow$ read_transition($TF$);
    add_to_adjacency_list($TGF, s', r, w, s$);
  end for
end while
```

(i.e., the state is *fresh*), then the state is given an index and written in the reachables file (if required) RF, whereas the corresponding transition is stored in the transitions file TF. If the state is a goal, its index is also written in the goals file GF, whereas if it is an error state the complete transition is written to the errors file EF. Finally, the state is enqueued in the disk queue Q to be expanded later by the algorithm.

It should be clear that this procedure is an revised version of the BUILDGRAPH of Procedure 3 improved supporting *hash compaction* and *disk storage*.

It is worth noting that *hash compaction* [158] is a state space reduction technique implemented in Murphi to reduce the size of each entry of the hash table making it more capacious. It associates to each system state a unique signature, storing in the hash table the state signature instead of the expanded state. In V-UPMurphi, when both *hash compaction* and *disk storage* are enabled, the algorithm uses the disk to store the system graph and the complete representation of each state encountered, whereas state signatures are stored in the hash table, to save space and enable hash compaction. However, the signature is not enough to compute the `next` function of a state since we need to access to the state variables. Hence, we modified the original Murphi hash compaction algorithm to store in the hash table both (1) the state signature and (2) a unique disk index which allows to directly access to the expanded state.

7.2.3 TRANSITION GRAPH GENERATION.

In this phase, the algorithm collects all the transition information generated by the model analysis and builds the inverted transition graph for the system. Indeed, the planning process requires to navigate the graph from the goals to the start states.

The Algorithm 6 reads the transitions file TF and writes each inverted transition to a set of adjacency lists, which are stored in the disk graph file TGF or in memory, if enough RAM is available. To this regard, the amount of RAM required to store the graph in memory is automatically evaluated by taking into account (1) the state space information collected in the previous phase, and (2) the size of the data structures needed to hold the memory graph.
The second column of Figure 7.2 summarises the memory and disk data structures used by this phase.

7.2.4 OPTIMAL PATHS CALCULATION.

At this point, the algorithm has all the information needed to decide the action to take in each systems state, e.g., calculate the (optimal) control paths for each state that can reach a goal. The algorithm can be configured (via the user-specified options) to choose any feasible action, or the action with minimum/maximum weight.

Procedure 7 COMPUTE OPTIMAL PATHS

```
1:  $Q' \leftarrow \emptyset$;
2:  **for** $(i = 0$ **to** number_of_states$)$ **do**
3:    chosen_edge[i]$\leftarrow$ $null$;
4:    distance[i] $\leftarrow \infty$;
5:  **end for**
6:  **while** $(TGF$ is not completely read$)$ **do**
7:    $s \leftarrow$ read_state_index($GF$);
8:    enqueue($Q', s$);
9:  **end while**
10: **while** $(Q'$ is not empty$)$ **do**
11:   $s \leftarrow$dequeue($Q$);
12:   **for** $(r, w, s') \in$adjacency_list($TGF, s$) **do**
13:     **if** (chosen_edge[$s'$] $= null \vee$
          distance[$s'$] $>$ distance[$s$] $+ w$) **then**
14:       **if** (chosen_edge($s'$) $= null$) **then**
15:         enqueue($Q', s'$);
16:       **end if**
17:       chosen_edge[$s'$] $\leftarrow (r, w, s)$;
18:       distance[$s'$] $\leftarrow$ distance[$s$] $+ w$;
19:     **end if**
20:   **end for**
21: **end while**
22: **for all** ($s$ | chosen_edge[$s$] $\neq null$) **do**
23:   write_action($AF, s$,chosen_edge[$s$]);
24: **end for**
```

Procedure 8 PLAN GENERATION

```
1: **for** $(s \in SF)$ **do**
2:   $plan \leftarrow \emptyset$;
3:   **while** (has_action($AF, s$)) **do**
4:     $(r, w, s') \leftarrow$read_action($AF, s$);
5:     append_to_plan($plan, s, r$);
6:     $s \leftarrow s'$;
7:   **end while**
8:   write_plan($PF, plan$);
9: **end for**
```

The process is implemented as shown in Algorithm 7 and uses the memory and disk data structures shown in the last column of Figure 7.2. Also in this case the procedure is an revised version of the UPLAN_GENERATION of Procedure 4 which now supports *hash compaction* and *disk storage*.

The disk queue Q' is initialised with the goal states found in the goals file GF, then the algorithm traverses the transition graph TGF generated by the previous phase using a suitably modified version of the Dijkstra algorithm. The chosen edges and the corresponding weights are stored in memory and, when the process is complete, the whole structure is written to the actions file AF.

7.2.5 Plan Generation.

If the user requires the generation of plans (and not a simple control table), this phase accesses the data startstates SF and actions AF files and writes in the plan file PF the paths starting from the user-specified start states (or from all the states, if a universal plan is required) as illustrated by Algorithm 8.

Finally, the planner reads the actions AF or plans PF file, depending on the kind of output requested by the user, and translates it in the appropriate output format, writing it to the output. This phase, as the previous one, works completely on disk data structures.

CHAPTER 8

PLANNING AND CONTROL CASE STUDIES

In this section we show a number of planning case studies for which the disk-based algorithm presented in Chapter 7 has been applied to synthesise plans and universal plan.

We first present some experimental results for two benchmark domains, i.e., the continuous version of the *Generator* domain as well as the *Cooling System* domain. We use these well-known case studies to show how the V-UPMurphi tool can synthesise plans and universal plan for domains having concurrency on processes and nonlinear dynamics.

Then, we present three significant case studies inspired by the real world specifications of complex systems, namely the *Engine Control of an Autonomous Planetary Lander*, the *Planetary Lander* and the *Batch Chemical Plant*. Complete details on these case studies, including the UPMurphi code generated from the PDDL+ domains, can be found on the UPMurphi web site [117], together with more complete experiment results. Moreover, when needed, we use the PDDL+ validator (VAL [96]) to *validate* the generated plans.

A preliminary versions of these results have been published in [56, 49, 50, 51, 52].

8.1 THE NONLINEAR GENERATOR DOMAIN

As a first example, we consider the continuous model of the *Generator* domain [95]. A generator is powered by a fuel tank with a limited capacity of 60 fuel units and consumes one fuel unit per second. During the generator activity (modelled by the the *consume* durative action), two fuel tanks of 25 fuel units each can be used to refuel it (through the *refuel* durative action). The refuelling activity is modelled as a durative action with variable duration (i.e., its duration must be decided by the planner) and is described by the Torricelli's law, which makes the system dynamics nonlinear. Moreover, the domain also involves concurrency, since the *consume* and *refuel* actions take place continuously and concurrently, and are modelled through continuous processes. The goal is to make the generator run for 100 seconds.

The PDDL+ model of Figure 8.1 should help to clarify the matter. The durative action *generate* has a fixed duration of 100 and requires that the generator fuel level is always positive. Its behaviour affects the fuel level of the generator. Differently, the durative action *refuel* requires to specify a generator `?g` and a fuel tank `?t` whilst duration depends to the tank `?t` volume. Its continuous effect is twofold: on one side it increases the generator fuel level and the refuel time, on the other side it decreases the tank fuel volume. It is worth noting that PDDL+ does not support the `sqrt` function, which is required to compute the initial tank's volume. Nevertheless, according to [95] we use a linear function of time to supply the square root of the initial volume of the tank.

State space size	10^{18}
Reachable states	$29,119,047$
Generated plans	$126,553$
Total synthesis time (sec)	$1,430.11$

Table 8.1: Universal Plan statistics for the generator domain.

The PDDL+ domain and problem of the generator (as shown in Figure 8.1) as given as input to V-UPMurphi in order to synthesize a Universal Plan having one generator and *two* different fuel tanks. Table 8.1 summarizes the results of the universal planning process. The final universal plan contains $126,553$ plans, which is a small fraction of the near 30 million states that the system can reach, showing that there are many situations in which the goal cannot be achieved (i.e., a plan cannot be devised by the planner).

An example of plan is given in Figure 8.1 while its VAL validation report is shown in Figure 8.2. Roughly speaking, VAL validates a plan with respect to the domain and problem file. It executes the plan by verifying if the goal is reached and if the plan obeys the domain constraints (e.g., the precondition of an action is satisfied, or the duration of a durative action is hold). Moreover, VAL is able to generate a graphical representation of the trend of each continuous variable (on the y-axis) with respect to the time (on the x-axis). In Figure 8.2 the trend of tanks volumes is depicted. We can note that during the first 59 seconds the fuel level decreases linearly since no refuel action is performed. Then the generator is refuelled using tank 1 in the time interval $[59,84]$ and tank 2 in the time interval $[75,87]$ (thus in the time interval $[75,84]$ the generator is refuelled using both tanks). Finally, the generator uses the remaining fuel to complete the task.

8.2 The Cooling System Domain

In this case study we considered a classical open thermodynamic system that generates energy, part of which is lost in friction and hydraulic losses and transformed into heat. The system shown in Figure 8.3 is composed by an external part, where a pump pours water continuously at a given rate into two hoses, and an internal part, composed by three water tanks which leak water at constant rate. The water passes through the pump and is poured by the hoses into two of the tanks at a time. We assume that hoses can be instantaneously

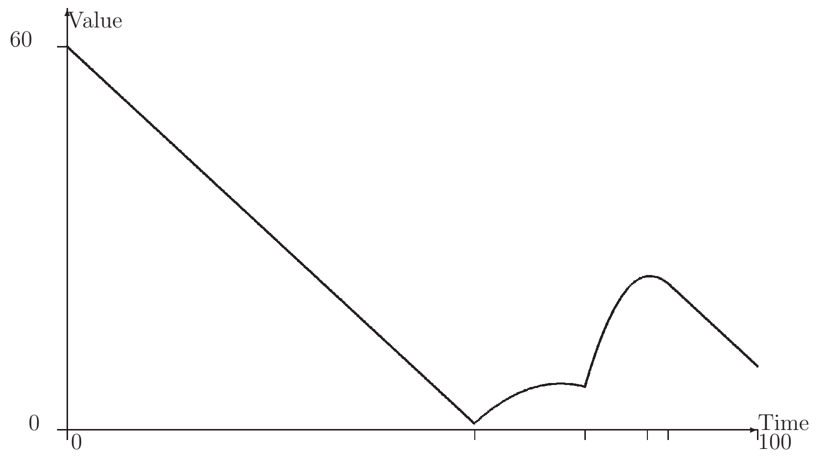
PDDL+ generator domain

```
(define (domain generator2)
(:requirements :fluents :durative-actions
  :duration-inequalities)
(:types gen tank)
(:predicates (refueling ?g - gen ?t -
tank)
  (generator_ran ?g - gen))
(:functions (tank_fuel_level ?t - tank)
  (gen_fuel_level ?g - gen)
  (flow_constant ?t - tank)
  (refuel_time ?t - tank)
  (capacity ?g - gen)
  (sqrtvolinit ?t - tank)
  (sqrtvol ?t - tank))

(:durative-action generate
 :parameters (?g - gen)
 :duration (= ?duration  100)
 :condition (over all (> (gen_fuel_level
 ?g) 0))
 :effect (and (decrease (gen_fuel_level ?
 g) (* #t 1))
       (at end (generator_ran ?g))))

(:durative-action refuel
 :parameters (?g - gen ?t - tank)
 :duration  (<= ?duration  (* (/ 1 (
 flow_constant ?t)) (sqrtvolinit ?t)))
 :condition (and (at start (not (
 refueling ?g ?t )) )
   (over all (< (gen_fuel_level ?g) (
   capacity ?g))))
 :effect (and (at start (refueling ?g ?t)
 )
  (at start (assign (refuel_time ?t) 0))

  (at start (assign (sqrtvol ?t) (
  sqrtvolinit ?t)) )

  (increase (refuel_time ?t) (* #t 1))

  (decrease (sqrtvol ?t) (* #t (
  flow_constant ?t)) )

  (decrease (tank_fuel_level ?t) (* #t (*
   (* 2 (flow_constant ?t)) (- (
  sqrtvolinit ?t) (* (flow_constant ?t) (
  refuel_time ?t)))) ))

  (increase (gen_fuel_level ?g) (* #t (*
  (* 2 (flow_constant ?t)) (- (
  sqrtvolinit ?t) (* (flow_constant ?t) (
  refuel_time ?t)))) ))

  (at end (not (refueling ?g ?t)))

  (at end (assign (sqrtvolinit ?t) (
  sqrtvol ?t)) )))
)
```

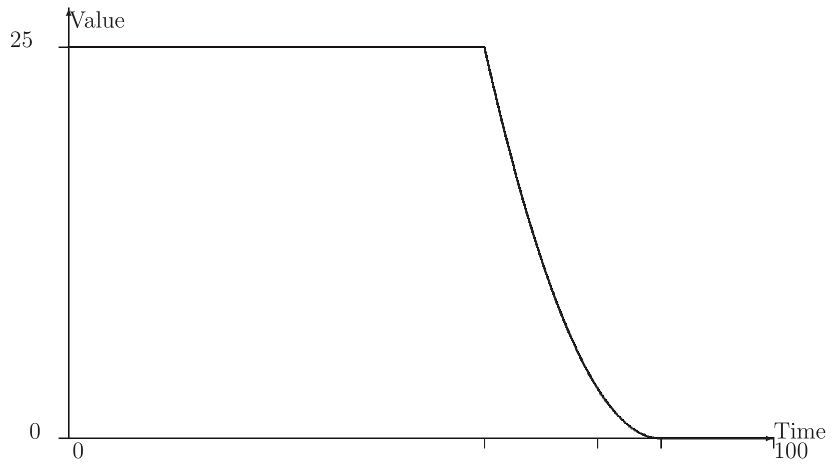
PDDL+ generator problem and plan

```
(define (problem run-generator2)
    (:domain generator)
    (:objects generator - gen tank1
     tank2 - tank)
    (:init
      (= (gen_fuel_level generator)
       60)
      (= (capacity generator)  60)
      (= (tank_fuel_level tank1) 25)
      (= (sqrtvolinit tank1) 5)
      (= (flow_constant tank1) 0.2)
      (= (tank_fuel_level tank2) 25)
      (= (sqrtvolinit tank2) 5)
      (= (flow_constant tank2) 0.4))
     (:goal (generator_ran generator))
     (:metric minimize (total-time))
)

; plan

000:  (generate generator) [100]
059:  (refuel generator tank1) [25]
075:  (refuel generator tank2) [12]
```

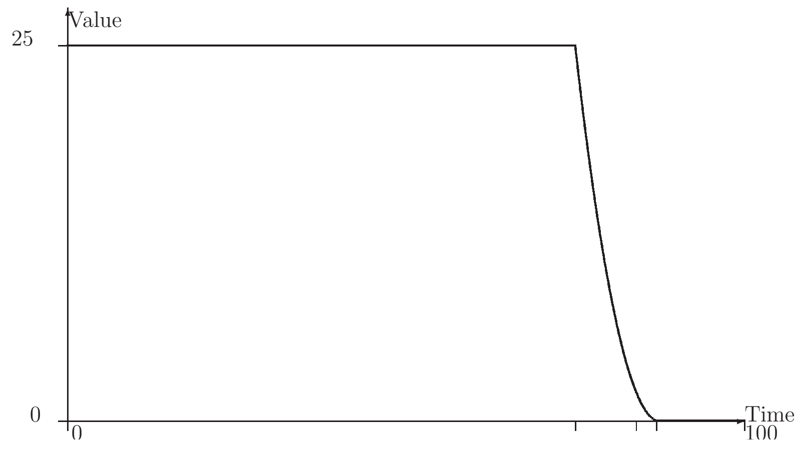
Figure 8.1: The PDDL+ continuous generator domain



(a) Fuel level for generator



(b) Fuel level for tank 1



(c) Fuel level for tank 2

Figure 8.2: VAL's Validation report for a single plan execution of the generator domain

repositioned on any tank. Moreover, we also consider that the water temperature raises when it passes through the pump, since it is heated by the pump engine.

The goal is to keep the amount of water in each of the three tanks above r_1, r_2 and r_3 liters, respectively, for 60 seconds. Moreover, we want the temperature T of the water passing through the pump to stay below 65 degrees.

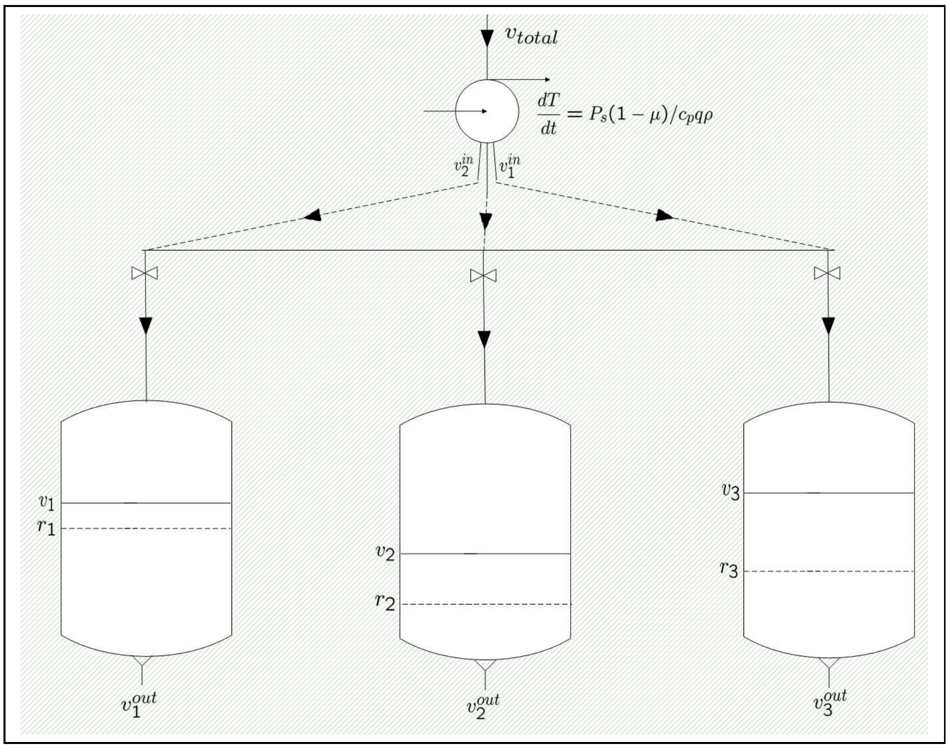


Figure 8.3: A graphical representation of the cooling system domain

Let v_i, with $i \in W = \{1,2,3\}$, denote the volume of water in Tank i and $v_i^{out} > 0$ denote the flow of water out of Tank i. Moreover, let v_j^{in}, with $j \in H = \{1,2\}$, denote the flow of water introduced into the system through hose j, where $v_{total} = v_1^{in} + v_2^{in}$ denotes the water flow that passes through the pump.

The system is equipped with a controller that switches a hose to Tank i whenever $v_i \leq r_i$. The boolean variable $filling_{i,j}$ is true when tank i is filled through hose j. Therefore, the variation of the volume of the water in tank i is given by the following equation:

$$\frac{dv_i}{dt} = \begin{cases} v_j^{in} - v_i^{out} & \text{if } \exists j \in H | filling_{i,j} = 1 \\ v_i^{out} & \text{otherwise} \end{cases}$$

Finally, the water temperature rising can be computed as follows:

$$\frac{dT}{dt} = P_s(1-\mu)/c_p q \rho$$

where the constant values are given in Table 8.2.

q	volume flow through the pump (m^3/s)	0.0006
P_s	brake power (kW)	0.095
μ	pump efficiency	0.05
c_p	specific heat capacity of the fluid $(kJ/kg\ °C)$	4.2
ρ	fluid density (kg/m^3)	1,000

Table 8.2: Cooling system constants

In the initial state of the system, the tanks are correctly filled and the water temperature respects the given constraint. However, the pump takes time to operate at full capacity, thus during the plan execution it increases its power by Δ_{power}, generating more heat. In this case, the planner may decide to increase the pump flow v_{total} by Δ_{rate} in order to mitigate the temperature rising (since the water flow cools down the pump), thus increasing also each v_i^{in} by $\frac{v_i^{in}}{v_{total}} \cdot \Delta_{rate}$, which may violate the constraint on the tank water level. On the other hand, the planner may leave v_{total} unchanged, risking to violate the maximum water temperature constraint.

Figure 8.4 shows the PDDL+ model of the cooling system domain, composed by a durative action *fill* that, given a tank `?t` and hose `?h`, starts the filling action of `?t` through `?h`, which is performed by the process *fill_tank*. Similarly, the tank leaking is modeled through process *leak_tank*. Note that the two processes may affect *concurrently* the same tank. The event *over-range* is used to invalidate plans in which exists at least one tank where $v_i < r_i$ or $v_i > c_i$, while event *overflow* invalidates all plans for which a hose does not fill any tank. The event *pump-danger* is triggered when the temperature of the water passing through the pump is greater than the *danger_level*.

The process *system_activity* is used to measure the time elapsed since the beginning of the first *fill* action. Indeed, after a given amount of execution, the event *power-increasing* is triggered, increasing the pump's power consumption p_s by Δ_{power}. The effect of such event is to allow the execution of the action *increase_rate*, which in turn increases the pump flow v_{total} by Δ_{rate}, modifying the water flow in the hoses and changing the dynamics of the system from linear to nonlinear.

State space size	10^{21}
Reachable states	$33,059,357$
Generated plans	$17,188,665$
Total synthesis time (sec)	$1,430.11$

Table 8.3: Cooling system universal plan generation statistics.

We used V-UPMurphi to generate the universal plan that controls the system activity for exactly one minute, starting from the initial condition shown in Figure 8.5. The results in Table 8.3.

In Figure 8.6 we show an example of validation report for a complete plan starting from

PDDL+ durative action

```
(define (domain cooling_system)
(:types tank hose pump)
(:predicates (fail)
  (system_start)
  (filling ?t - tank ?h - hose)
  (busy ?h - hose)
  (warning))
(:functions (v ?t -  tank)
  (v_out ?t - tank)
  (c ?t - tank)
  (v_in ?h - hose)
  (r ?t - tank)
  (system_counter)
  (temp ?p - pump)
  (p_s) (mu) (c_p) (q)
  (rho)
  (delta_rate)
  (delta_power)
  (plan_length)
  (danger_level))
(:durative-action fill
 :parameters (?t - tank ?h - hose)
 :duration (>= ?duration  0)
 :condition  (at start (not(busy ?h)))
 :effect (and
  (at start (busy ?h))
  (at start (syste_start))
  (at start (filling ?t ?h))
  (at end (not(filling ?t ?h)))
  (at end (not(busy ?h)))
 )
)
(:action increase_rate
 :parameters (?h1 ?h2 - hose)
 :precondition (warning)
 :effect (and
  (increase (v_in ?h1)
  (* 1000 (* (delta_rate) (/(v_in ?h1)(q)
   ))))
  (increase (v_in ?h2)
  (* 1000 (* (delta_rate) (/(v_in ?h2)(q)
   ))))
  (increase (q)
    (delta_rate)))
)
```

PDDL+ processes and events

```
(:event over-range
 :parameters (?t - tank)
 :precondition (or
  (< (v ?t) (r ?t))
  (> (v ?t) (c ?t)))
 :effect (fail)
)
(:event overflow
 :parameters (?h1 ?h2 - hose)
 :precondition (and
  (system_start)
  (or (not (busy ?h1))
  (not (busy ?h2))))
 :effect (fail)
)
(:event pump-danger
 :parameters (?p - pump)
 :precondition
  (> (temp ?p)
  (danger_level) )
 :effect (fail)
)
(:event power-increasing
 :parameters ()
 :precondition (and
  (>= (system_counter) (/ (plan_length
   ) 2))
  (not(warning)) )
 :effect (and (warning)
  (increase (p_s) (delta_power)))
)
(:process fill_tank
 :parameters (?t - tank
  ?h - hose)
 :precondition (and
  (filling ?t ?h)
  (system_start))
 :effect (increase
  (v ?t) (* #t (v_in ?h)))
)
(:process leak_tank
 :parameters (?t - tank)
 :precondition (and (system_start))
 :effect (decrease (v ?t)
  (* #t  (v_out ?t)))
)
(:process system_activity
 :parameters ()
 :precondition (system_start)
 :effect (increase (system_counter)
  (* #t 1))
)
```

Figure 8.4: The PDDL+ cooling system domain

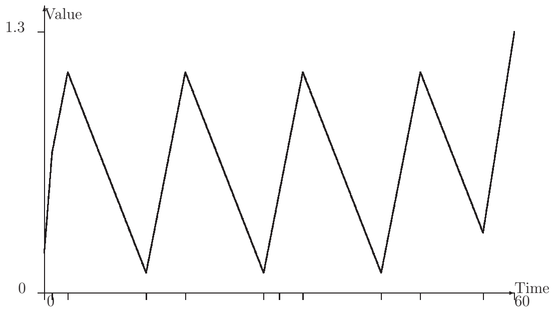
PDDL+ problem

```
(define (problem cooling_system_1)
(:domain cooling_system)
(:objects tank1 tank2 tank3 - tank
  hose1 hose2 - hose pump1 - pump)
(:init
  (= (v tank1) 0.2) ;liters
  (= (v tank2) 0.6)
  (= (v tank3) 0.9)
  (= (c tank1) 1.5) ;liters
  (= (c tank2) 1.5)
  (= (c tank3) 1.5)
  (= (r tank1) 0.1) ;liters
  (= (r tank2) 0.2)
  (= (r tank3) 0.2)
  (= (v_out tank1) 0.1) ;liters
  (= (v_out tank2) 0.3)
  (= (v_out tank3) 0.2)
  (= (v_in hose1) 0.3) ;liters
  (= (v_in hose2) 0.3)
  (= (system_counter) 0)
  (= (q) 0.0006) ; the sum of v_in in m^3
  (= (p_s) 0.13) ; kW
  (= (mu) 0.05)
  (= (c_p) 4.2 ) ; kJ/kg°C
  (= (rho) 1000 ) ; Kg/m^3
  (= (delta_power) 0.19) ; kW
  (= (delta_rate) 0.00005) ; in m^3
  (= (temp pump1) 60)
  (= (plan_length) 60)
  (= (danger_level) 65)
  (system_start)
  (not(fail))
)
  (:goal (and
    (not(fail))
    (= (system_counter) (plan_length))) )
  (:metric minimize (total-time)))
```

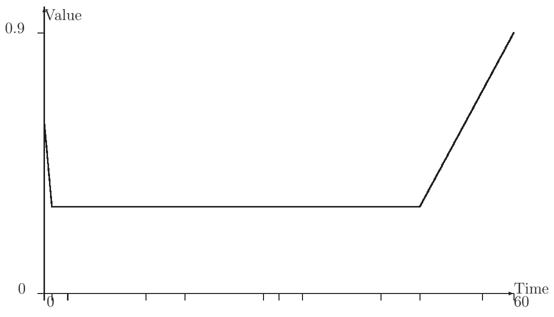
PDDL+ plan

```
000:(fill tank1 hose1) [001]
000:(fill tank1 hose2) [003]
001:(fill tank2 hose1) [059]
003:(fill tank3 hose2) [010]
013:(fill tank1 hose2) [005]
018:(fill tank3 hose2) [010]
028:(fill tank1 hose2) [005]
033:(fill tank3 hose2) [010]
043:(fill tank1 hose2 ) [005]
048:(increase_rate hose1 hose2)
048:(fill tank3 hose2 ) [008]
056:(fill tank1 hose2 ) [004]
```

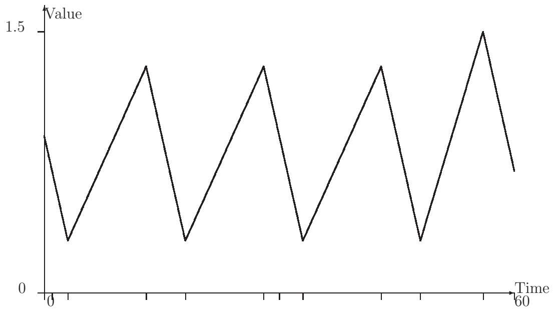
Figure 8.5: The PDDL+ cooling system problem and one of the devised plans



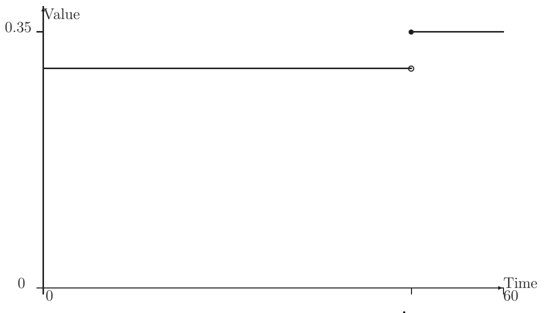
(a) Trend for variable v_1



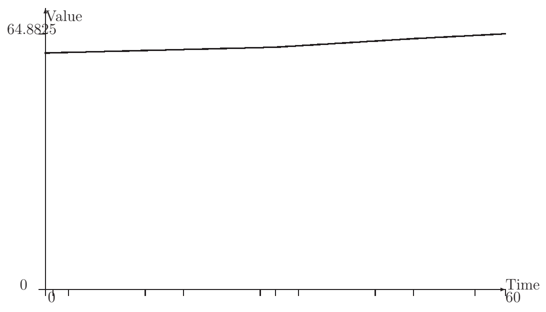
(b) Trend for variable v_2



(c) Trend for variable v_3



(d) Trend for variables v_j^{in}



(e) Trend for variable *temperature*

Figure 8.6: Validation report for a single plan execution of the cooler system domain.

the start state given in Figure 8.5. In particular, Figures 8.6a, 8.6b, 8.6c describe the amount of water v_i in each tank i, Figure 8.6d shows the evolution of v_j^{in} for each hose j, and Figure 8.6e the water temperature rise.

8.3 THE ENGINE CONTROL OF AN AUTONOMOUS PLANETARY LANDER

This section presents a case study where planning is applied to automatically control the engine of an autonomous vehicle during a planetary exploration mission.

Autonomous planetary vehicles, commonly known as rovers, are a great challenge in the field of autonomous vehicles, since they have often to take actions on a hazardous ground with narrow time and energy consumption constraints. Rovers operating on distant planets may receive commands from Earth operators only once per day, and during the remaining time they have to perform a specific mission, which may include moving to a specific place, position some instruments, take measures, etc [87, 159].

Usually, rover activities are converted on the ground into a detailed plan that, once generated and uploaded to the vehicle, drives it for the rest of the mission. Therefore, planning for this kind of autonomous vehicles should be very precise and take into consideration many factors [167].

Many rover activities begin with a movement that places it in a specified location. Thus, independently from the nature of the rover's mission, reaching the activity location is the first goal to achieve, and it must satisfy two main constraints: energy and time consumption. This is the case of the *Engine Control of an Autonomous Planetary Lander* where we show how V-UPMurphi can be used to generate optimal plans to control a *rover's engine*, in order to move it for a specific distance in the least possible time, while satisfying a set of technical constraints and trying to save energy.

In the presented case study, the rover dynamics and behaviour, including some common technical constraints, have been modelled through general equations, that may apply to a wide range of vehicles. Plans have been optimized to minimise energy and time requirements, and the given minimal battery charge is always preserved. Therefore, the results are quite realistic.

The rover can be naturally modelled as a hybrid system, with several nonlinear characteristics. Thus, we have a dynamics very hard to compute, which makes planning quite difficult.

8.3.1 Rover Specification

The rover model used in our case study is based on the Mars exploration rover described in [120].

In general, an exploration rover moves on the planet surface to observe different phenomena and/or try some experiments. The rover can recharge its batteries through a solar panel, but recharge cannot take place continuously, and the energy from the panels is not enough to directly power the rover. Therefore, it must minimize the energy consumption in order to have always enough battery charge for the next activity.

Moreover, the rover has limited communication and computation resources, so it must be programmed with a detailed plan of activity and then left operating, without any chance to recover from an error or recompute its mission. If something wrong or unexpected happens, the best that the rover can do is to stop, reset and wait for the next Earth connection to get new instructions.

The plan we want to generate does not address the actual *route* of the rover, but *controls the vehicle engine and instruments* during the route itself. Routing is a different problem, so just we assume that a (possibly straight) route of length d_{final} has been separately planned and will be used to control the steering of the rover wheels.

When moving, the rover is subject to friction and drift due to the - often unpredictable - ground characteristics. Thus, every d_{max} meters, it has to stop for t_c seconds to look at its actual position and conditions, before starting again to move. These frequent stops may also be useful to ensure a proper cooling of the rover wheels and instruments, if moving in a hot environment. For sake of generality, in the following we shall call these stops “cooling tasks”. However, we assume that the route duration be less or equal to t_{max} seconds, since the overall rover mission should not exceed a reasonable limit.

The rover has a base energy consumption g_s Joule/second, used to power its CPU.

The energy (expressed in Joule/second) required to move the rover with speed v and acceleration $\dot{v}$ can be evaluated by applying the general function f of Equation 8.1, where m is the vehicle mass and fa is its frontal area (see [162] for details).

$$f(v,\dot{v}) = \left(\frac{1}{2}\cdot\rho\cdot v^2\cdot Cd\cdot fa + m\cdot g\cdot\left(Crr+\frac{\dot{v}}{g}\right)\right)\cdot v \tag{8.1}$$

In the equation, constants ρ, g indicate the planet air density and its gravitational constant, respectively, whereas Cd and Crr are the drag and rolling coefficients of the rover.

Finally, the cooling tasks require a constant energy of g_c Joule/second.

The rover dynamics (i.e., the covered distance d, the speed v and the acceleration $\dot{v}$) is given by Equation 8.2.

$$\begin{aligned} \frac{\partial v}{\partial t} &= a(t) - \mu \cdot g \\ \frac{\partial d}{\partial t} &= v(t) \end{aligned} \tag{8.2}$$

where $a(t)$ is the acceleration given by the rover motor at time t and μ is the kinetic friction coefficient for the rover wheels.

We assume that, in each communication session, the Earth control sends to the rover a plan to drive it to the next place, and the commands needed to start the corresponding activity. Such plan consists of a sequence of actions, to be performed at 1 second intervals, chosen from the set $A = \{$*accelerate*, *decelerate*, *continue* (moving at constant speed), perform a *cooling* task$\}$.

The plan must obey the following constraints:

- the rover must not exceed the speed of v_{max};
- the rover must stop every d_{max} to perform a cooling task;
- the rover must stop after d_{final} (to start the activity) with a residual battery charge not lower than c_{min};
- the rover route must not require more than t_{max} seconds.

In particular, we must ensure that, after moving to the given location, the rover has still enough battery charge available for its activity.

Finally, the plan must drive the vehicle to its goal as soon as possible, since saving time allows the rover to complete more activities before its life terminates.

8.3.2 Rover Modelling

The dynamics and constraints given above have been first modelled as a hybrid automaton, shown in Figure 8.7. The state of the automaton is $s = (x, q) \in S$, where $q \in \{$*stopped*, *running*, *braking*, *cooling*, *engine blown*, *no energy*$\}$ and $x = (d, a, v, T, T_c)$.

The rover is initially in a *stopped* state, where the only energy consumption is given by g_s. When started, the rover enters the *running* state and moves as described by Equation 8.2 while its energy consumption is increased by the value given by Equation 8.1. The vehicle can accelerate and decelerate with steps of $1.5cm/s^2$. After d_{max} meters, the vehicle starts *braking* and, once stopped, it begins the *cooling* phase, with the corresponding energy consumption. After 6 seconds of cooling (T_c in the automaton), the vehicle restarts and continues in the *running* state.

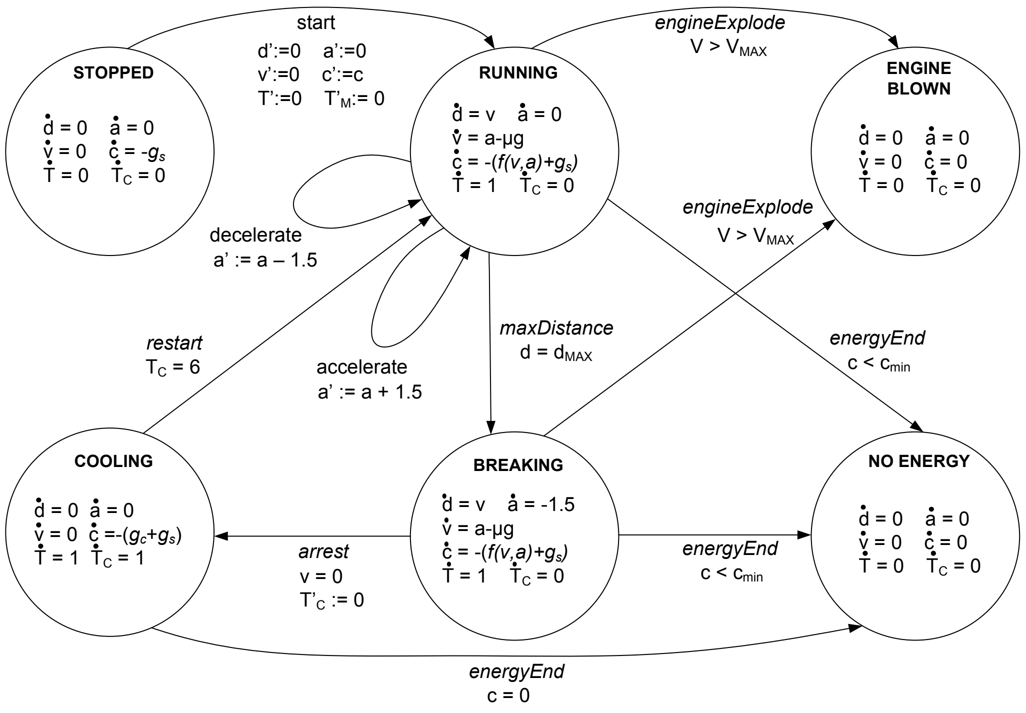


Figure 8.7: Hybrid automaton for the Control of Autonomous Planetary Lander case study

The automaton also shows two possible failure conditions: if the rover moves faster than the max allowed speed v_{max}, its engine blows up (*engine blown* state): in this case, the entire mission could fail. On the other hand, if the consumed energy exceeds the limit c_{min}, the rover stops (*no energy* state), using the residual energy to wait for Earth instructions.

We fixed the model constants to the values given in Table 8.4, most of which are obtained from rover specifications like [159] and [87]. Note that we assume that the rover operates on the Mars surface.

Finally, according to Definition 8, we evaluate the cost of the generated plan through the function $C(s_i, a_i)$ defined as in Equation 8.3.

$$\begin{cases} \frac{{g_s}^2}{t_{max}-i} + C_a(a_i) & \text{if } q_i = stopped \\ \frac{(g_s+g_c)^2}{t_{max}-i} + C_a(a_i) & \text{if } q_i = cooling \\ 0 & \text{if } q_i \in \left\{ \begin{array}{l} no\ energy, \\ engine\ blown \end{array} \right\} \\ \frac{(g_s+f(v_i,\dot{v}_i))^2}{t_{max}-i} + C_a(a_i) & \text{otherwise} \end{cases} \tag{8.3}$$

where $s \in S, a \in A$. Here, $C_a = 0$ since all the actions are instantaneous and do not require

Table 8.4: Constant values for the rover model

ρ	Air density	0.1 Kg/m^3
g	Gravitational acceleration	3.8 m/s^2
m	Vehicle mass	71.73 Kg
μ	Kinetic friction coefficient	0.8
c_{max}	Initial battery charge	$18,000$ C
c_{min}	Min final battery charge	$17,000$ C
v_{max}	Max speed	10 cm/s
a_{max}	Max acceleration	5 cm/s^2
g_s	CPU energy requirements	25 J/s
t_c	Cooling duration	6 s
d_{max}	Distance between coolings	1.30 m
g_c	Cooling energy requirements	10 J/s
d_{final}	Final distance	2 m
t_{max}	Max plan duration	60 s

energy.

This definition of C allows one to perform optimization on both energy and time, as required, still giving more importance to the energy component. Indeed, usually the mission could be accomplished even if it requires some seconds more than the planned limits, whereas running out of battery charge could lead to dangerous failures.

The resulting model has been translated to a FSS, encoded in the CMurphi description language, with the same state variables and transition function of the hybrid automaton in Figure 8.7. In this phase, the continuous state variables have been suitably discretised: in particular, we applied an approximation of 0.1 to all the variables, thus the total number of different states of the FSS is $2.2 \cdot 10^{13}$.

8.3.3 UPMurphi Model

The UPMurphi rover model, we applied an approximation of 0.1 to all the variables, and introduced the $v_{safemax} \leq v_{max}$ constant as the actual maximum speed. This gives us a chance to set a further safety threshold on the speed, to prevent an engine blow due to approximation errors. On the other hand, the journey time will be measured in seconds, since it is a reasonable update interval for the rover engine status. It is worth noting that, with the given discretisation, the total number of different states of the FSS is $2.2 \cdot 10^{13}$.

Figure 8.8 shows the resulting UPMurphi code, where for sake of simplicity we omit the declaration of constants and state variables.

The start state of the model, *stopped*, describes the corresponding initial state of the hybrid automaton, i.e., fixes the initial conditions of the rover. Then, the *start* rule initiates the rover movement by setting the running variable to true.

The other five model rules, namely *accelerate*, *decelerate*, *running*, *braking* and *cooling*, model the main transitions and states of the automaton. In particular, *accelerate* and *decelerate* update the acceleration variable as described by the corresponding automaton transitions. These rules have a null duration and weight, according to the hybrid automaton semantics, since they represent instantaneous updates.

On the other hand, the *running*, *braking* and *cooling* rules have duration 1, since they model the changes in the rover state (i.e., speed, distance and battery charge) during a time step of one second. Such updates are actually performed by the *running_status_update*, *braking_status_update* and *cooling_status_update* procedures, respectively, which concentrate the update logic found in the entire automaton, i.e., the updates specified on the *maxDistance*, *arrest* and *restart* transitions and the ones contained in the running and cooling states. The status update procedures, in turn, compute some values through external C functions (e.g., *update_c_cooling*) that are used to evaluate the complex expressions. Moreover, the external functions *cost_moving* and *cost_cooling* are used to dynamically calculate the weight of each rule, as defined by the cost function shown by Equation 8.3. The invariants *engineExplode* and *energyEnd* model the homonymous transitions that lead, in the automaton, to error states (*engineBlown* and *noenergy*, respectively). These states are not modelled here, since the planner automatically detects as errors all the states that violate an invariant. Finally, the goal construct is used to declare the success condition of the model, i.e., when the rover completes successfully its journey.

8.3.4 Planning

To build the optimal plan, the FSS was given in input to V-UPMurphi, which generated 939,477 reachable states in 2,257 seconds, with a peak memory requirement of 500 MB. Note that the reachability analysis performed by the tool allowed us to consistently prune the system state space, as reported in Table 8.6. The resulting plan is described in Table 8.5.

The table reports, for each second (which is the plan sampling time, as discussed earlier) the model rule (with respect to the code in Figure 8.8) chosen by V-UPMurphi. Thus, the rover starts its journey when *Start* is selected, moves when *Running* is selected, brakes when *Braking* is selected, increases or decreases its speed when *Accelerate* or *Decelerate* are selected, respectively, and performs a *Cooling* when the homonymous rule is chosen. Note that we may have more than one rule executed in a single time step, since some of them (namely, *Start*, *Accelerate* and *Decelerate*) have duration zero.

Model rules

```
startstate " stopped "
BEGIN
  a := 0.0; d := 0.0;
  v := 0.0; c := c_max;
  T_c := 0.0;
  cooling := false;
  braking := false;
  running := false;
END;

rule " start "
duration: 0;
weight: 0;
(!running & !cooling & !braking ) ==>
BEGIN
  running :=.true;
END;

rule " accelerate "
duration: 0;
weight: 0;
(running & !cooling & !braking ) ==>
BEGIN
  a := a + 1.5;
END;

rule " decelerate "
duration: 0;
weight: 0;
(running & !cooling & !braking) ==>
BEGIN
  a := a - 1.5;
END;

rule " running "
duration: 1;
weight: cost_moving();
(running & !cooling & !braking ) ==>
BEGIN
  running_status_update();
END;

rule " braking "
duration: 1;
weight: cost_moving();
(!running & !cooling & braking ) ==>
BEGIN
  braking_status_update();
END;

rule " cooling "
duration: 1;
weight: cost_cooling();
(!running & cooling & !braking) ==>
BEGIN
  cooling_status_update();
END;

invariant " engineExplode "
(!(running & v > v_safemax));

invariant " energyEnd " (!(c < c_min));

goal " success " (v = 0 & d = d_final);
```

Support procedures

```
procedure running_status_update();
BEGIN
  d := update_d(d,v,a);
  v := update_v (v,a);
  c := update_c(rho,v,m,g,a,h,f);
  -- maxDistance
  IF ((d = d_max) & (T_C = 0)) THEN
    braking := true;
    running := false;
  ENDIF;
END;

procedure braking_status_update();
BEGIN
  a := a - 1.5;
  d := update_d (d,v,a);
  v := update_v (v,a);
  c := update_c (rho,v,m,g,a,h,f);
  -- arrest
  IF (v=0 & a=0) THEN
    braking:= false;
    cooling:= true;
    T_c := 0;
  ENDIF;
END;

procedure cooling_status_update();
BEGIN
  T_c := T_c +1;
  -- cooling
  IF (T_c <= 6) THEN
    c := update_c_cooling(c,g,v,m);
  ELSE
  -- restart
    cooling := false;
    running := true;
  ENDIF;
END;
```

Figure 8.8: UPMurphi code for the Autonomous Planetary Vehicle case study.

Table 8.5: Optimal plan.

T(sec)	**Rule**	**T(sec)**	**Rule**	**T(sec)**	**Rule**
0	Start Accelerate Running	15	Running	30	Cooling
1	Accelerate Running	16	Running	31	Cooling
2	Running	17	Running	32	Cooling
3	Decelerate Running	18	Running	33	Cooling
4	Decelerate Running	19	Running	34	Cooling
5	Decelerate Running	20	Running	35	Cooling
6	Running	21	Running	36	Accelerate Running
7	Running	22	Accelerate Running	37	Running
8	Accelerate Running	23	Running	38	Decelerate Running
9	Running	24	Running	39	Decelerate Running
10	Running	25	Braking	40	Decelerate Running
11	Running	26	Braking	41	Decelerate Running
12	Running	27	Braking	42	Decelerate Running
13	Running	28	Braking		
14	Running	29	Cooling		

Table 8.6: Optimal plan statistics

Course length	43 s
Energy consumption	77.3 C
Residual battery charge	$17,922.7$ C
Time in *Stopped* state	1 s
Time in *Running* state	32 s
Time in *Braking* state	4 s
Time in *Cooling* state	6 s

It is worth noting that the plan optimization allowed us to save 922.7 C with respect to the required minimal battery charge, and 17 seconds with respect to the maximum allowed plan duration.

Finally, the generated plan has been further validated by simulating its execution on the rover model. The graphs in Figure 8.9 show the evolution of some important rover state variables during the simulation, which ends correctly after $d_{final} = 2\ m$. In particular, we can compare the battery discharge graph with the rover speed and acceleration during the entire course. Note that, in the highlighted cooling phase, the battery discharge rate is higher even if the vehicle is stopped, due to the instruments activation.

Another interesting plan analysis is given in Figure 8.10, where we plot the rover engine energy requirements, i.e., the value of f in Equation 8.1, and the value of the cost function $C(\pi)$during the plan evolution. The graph clearly shows that, as required, the plan cost is very tightly related to the energy consumption, since the battery charge is a critical resource, whereas the time has a considerably lower impact (for example, look at the small increment of the cost when the required energy is constant, between $T = 8$ and $T = 22$).

8.4 The Activity Planning for a Planetary Lander

In Section 8.3 we treated the problem of control the engine of an autonomous planetary lander. In this Section we address the activity planning problem for a planetary lander. Indeed, when the rover has reached its final position it needs to perform some observation tasks minimising power consumption, recharging the batteries during the sunlight. Moreover, the rover has to complete its task as soon as possible, since environmental conditions may quickly change, and in general a shorter task duration means that the rover will be able to perform more activities during the mission time. This is the case of *The Plantery Lander* domain, inspired by the Beagle2 Mars Lander [21] and proposed as a PDDL+ model in [75].

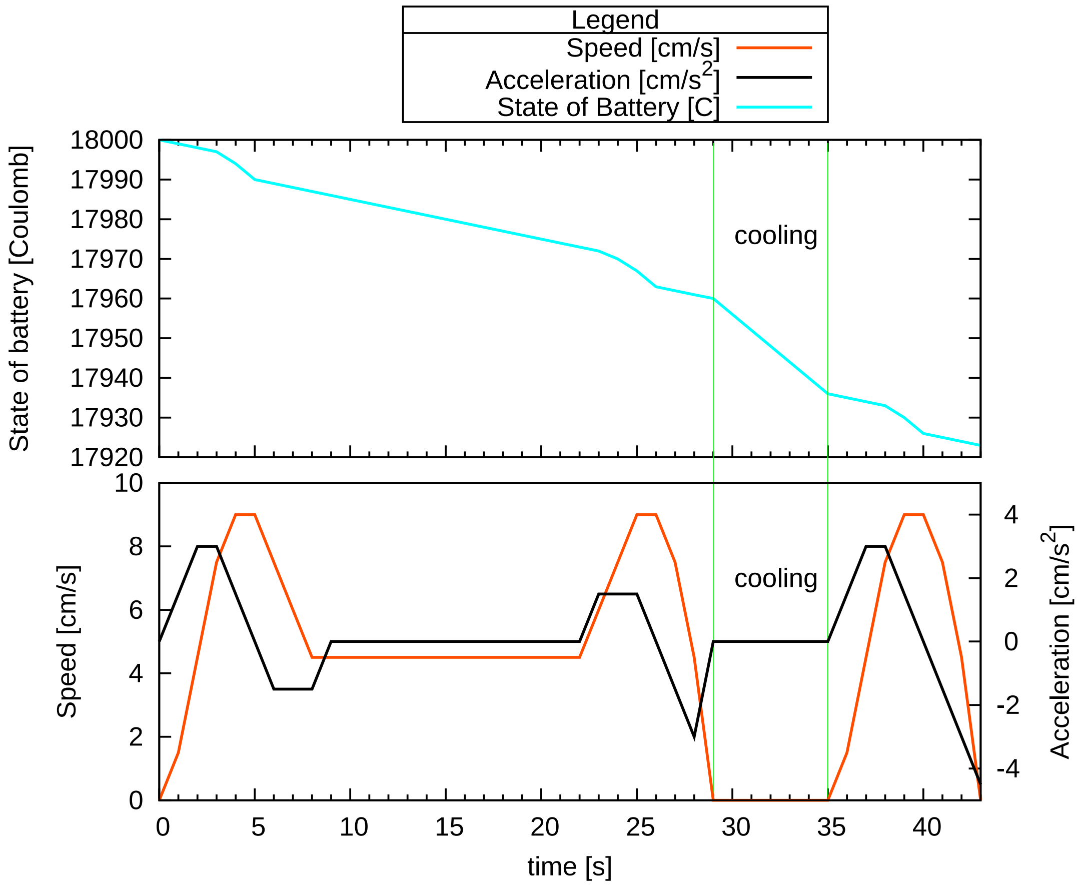


Figure 8.9: Optimal plan evolution: battery charge, speed and acceleration.

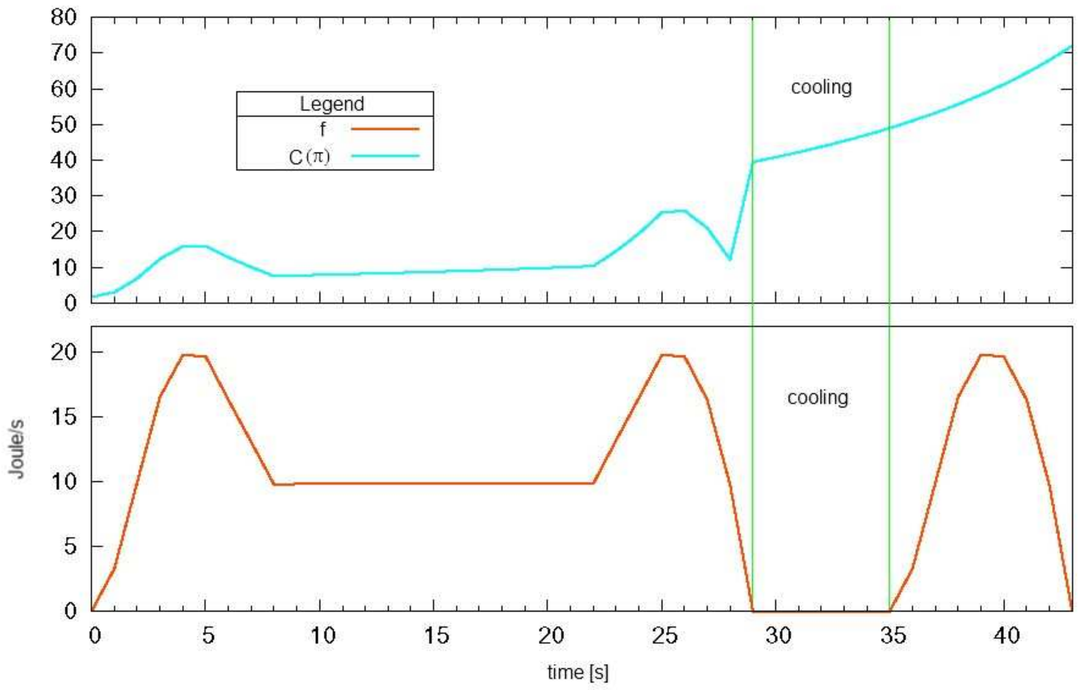


Figure 8.10: Optimal plan evolution: engine energy requirements and cost function.

The planetary lander domain and problem which we present are inspired by the specifications of the “Beagle 2” Mars probe [21], designed to operate on the Mars surface with tight resource constraints. In particular, we use the PDDL+ domain presented by [75], based on a simplified model of a solar-powered lander, the *Planetary Lander*. Table 8.7 shows an overview of the main domain elements with their preconditions.

Name	Type	Precondition
nightfall	Event	$(daytime >= dusktime)$ & day
daybreak	Event	$(daytime >= 0)$ & $\neg day$
charging	Process	$(supply >= demand)$ & day
discharging	Process	$(supply < demand)$
generating	Process	(day)
night-operation	Process	$(\neg day)$
fullprepare & prepareObs1 prepareObs2	Durative action	$\forall t \in ActionDuration$ $(battery >= safelevel)$
Observe1	Durative action	$readyForObs1$ & $\forall t \in ActionDuration$ $(battery >= safelevel)$
Observe2	Durative action	$readyForObs2$ & $\forall t \in ActionDuration$ $(battery >= safelevel)$

Table 8.7: A snapshot of the main PDDL+ domain elements for the planetary lander case study

Basically, the lander must perform two observation actions, called *Observe1* and *Observe2*. However, before making each observation, it must perform the corresponding preparation task, called *prepareObs1* and *prepareObs2*, respectively. Alternatively, the probe may choose to perform a cumulative preparation task for both observations by executing the single long action *fullPrepare*. The shorter actions have higher power requirements than the single preparation action.

The power needed to perform these operations comes from the probe solar panels. The energy generated by the panels (through the *generating* process) is influenced by the position of the sun, i.e., it is zero at night, rises until midday and then returns to zero at dusk. Power coming from the solar panels is also used to charge a battery (the *charging* process), which is then discharged to give power to the lander (the *discharging* process) when the panels do not produce enough energy (e.g., at night). Moreover, the probe must always ensure a minimum battery level to keep its instruments warm.

The state of charge of the battery is therefore an important variable to monitor. Unfortunately, it follows a complex curve, since the charge/discharge process is nonlinear, and has several discontinuities, caused by the initiation and termination of the actions. Indeed, Table 8.8 shows the set of ordinary differential equations that are used to recalculate the values of the state variables *soc* (state of charge) and *supply* (solar panel generation). The symbols used in the equations have the following meaning: $s = soc$, $h = supply$, $d = demand$, $r = charge\_rate$, $sc = solar\_const$ and $D = daytime$. The equations clearly show the nonlinear dynamics of the system.

Name	ODE
charging	$\frac{ds(t)}{dt} = [h(t) - d(t)] \cdot r \cdot (100 - s(t))$
discharging	$\frac{ds(t)}{dt} = -[d(t) - h(t)]$
generating	$\frac{dh(t)}{dt} = [sc \cdot D(t)] \cdot$ $\cdot [(D(t) \cdot ((4 \cdot D(t)) - 90))] + 450$

Table 8.8: PDDL+ events and processes for the planetary lander case study, with associated ordinary differential equations

Obviously, the problem here is to find the best correct sequence of actions to achieve the probe goal in the shortest time possible, starting from any reasonable initial configuration. For sake of brevity, here we do not show the PDDL+ problem domain, which can be read in [75].

8.4.1 Domain Specification

The start state cloud for the universal planning algorithm was selected by taking into account a set of reasonable configurations of the state variables *soc* and *daytime*. Note that it is realistic to consider *only* these parameters, since they define the environmental conditions to which the lander will be subject at the beginning of its mission. All the other domain parameters were fixed to the values inferred by looking at [21].

In particular, we suppose that the rover landing hour may be between 0 and 8, that corresponds to the central daylight hours in Martian time (the rover is supposed to land in this range of hours, since they offer the best possible starting conditions). On the other hand, since the battery is not used before landing, and its self-discharge rate is minimal, we can safely suppose that the initial battery state of charge will be between 90% and 100% with steps of 1%. Therefore, the start state cloud will be defined as the set $\{(s,d) | s \in [90\%, 100\%] \wedge d \in [0,8]\}$.

8.4.2 Universal Planning

Given the domain variables and their ranges as well as the time discretisation, we can easily calculate the space size of the system is about of 10^{24} states. Thanks to the reachability analysis, V-UPMurphi generated an optimal solution for the universal planning problem, starting from the given start state cloud, and visiting only a small fraction of the state space (i.e., 31 million of reachable states) in less than 40 minutes on a 2.2GHz CPU with 2 GB of RAM. The synthesis statistics are in Table 8.9.

Note that the first goal was found after 174 steps, but the synthesis was performed up to the fixed horizon of 200 steps, which is a reasonable upper bound for the lander activity completion (it represents about two Martian days).

State space size	10^{24}
Search depth limit	200 BFS levels
First goal reached after	174 BFS levels
Reachable states	$31,965,220$
Start states to goal	100%
States to goal (generated plans)	$5,309,514$
Forward analysis time	$1,969.3$ seconds
Plan generation time	296.51 seconds
Total synthesis time	$2,265.81$ seconds
Peak memory requirements (hash table)	1800MB

Table 8.9: Planetary lander universal plan generation statistics.

The generated solution contains more than 5 million plans, and thus it is able to bring to the goal more than 16% of the reachable states. Due to the exhaustive search performed by the tool, we can safely assert that, in the remaining 84% of the states, the lander could not complete its tasks and should therefore quit its mission or delay its initiation.

It is worth noting that, in this case, the use of the disk algorithm (as described in Chapter 7) has been used to synthesise the universal plan. To this regard, Table 8.11 shows the V-UPMurphi statistics about the disk usage.

Reachables File	1.7GB
Transitions File	552MB
The Graph File	323MB
The Action File	51MB

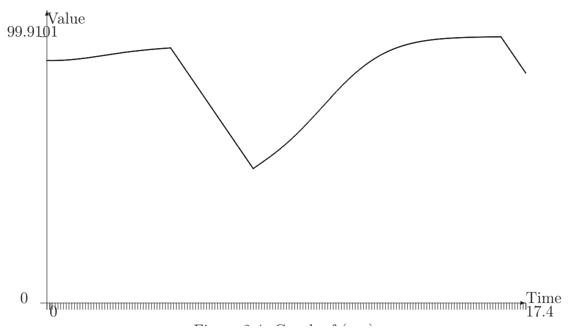
Table 8.10: Disk-based Algorithm statistics for the Planetary lander universal plan generation.

In Figure 8.11 we provide an extracted plan for planetary lander, whilst Figure 8.12 shows the VAL's plan validation report describing the evolution of variables *soc*, *supply*, *daytime*, and *demand* with respect to the time. In the reported example the lander starts its mission in the middle of the martian day with an almost full battery charge. In this case the planner preferred to performs the two preparation observations tasks instead of the *full-Prepare* task. The plan starts performing *prepareObs1* and *prepareObs2* which require a low energy demand (Figure 8.12d). During this phase, the lander can generates energy and recharge the battery (Figures 8.12a and 8.12b) whilst, during the martian night, all the energy is used to perform task and to heat instruments. Then, the plan devotes all the second mission day to perform the last observation task, using the daylight to generate energy.

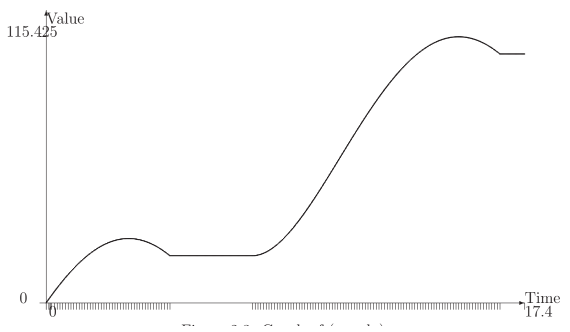
However, to further estimate the precision of the plans, we compared the variable values computed by VAL during the validation process with the corresponding values output by the UPMurphi plan synthesis process, computing the normalised root mean squared error

```
0.1:  (PrepObs2)  [1.5]
1.7:  (PrepObs1)  [1]
2.8:  (Obs2)  [7.5]
10.4:  (Obs1)  [7]
```

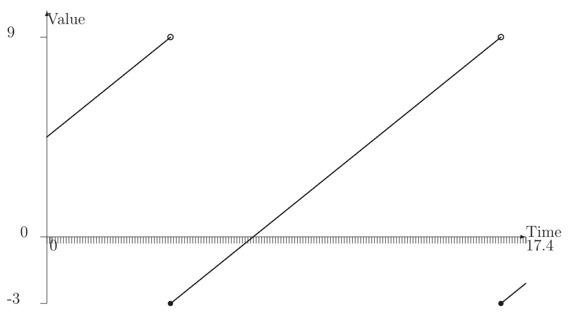
Figure 8.11: A plan for the planetary lander



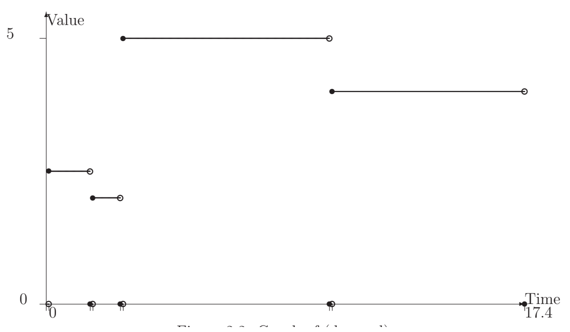
(a) Trend for variable *soc*



(b) Trend for variable *supply*



(c) Trend for variable *daytime*



(d) Trend for variables *demand*

Figure 8.12: Validation report for a single plan execution of the planetary lander domain.

(NRMSE), as shown in Table 8.11. The NRMSE is at most 2% in all the generated plans for the nonlinear variable *soc*, at most 0.6% for nonlinear variable *supply* and always zero for the linear variable *daytime* (not shown in the table). Nevertheless, the average NRMSE is small: 0.179% for *soc* and 0.742% for *supply*, respectively.

A repository of the generated PDDL+ problems and plans with validation reports can be found at [48].

	Min	Max	Avg
soc	0 %	0.625,392 %	0.179,329 %
supply	0 %	2.060,061 %	0.742,575 %

Table 8.11: Normalised root mean squared error for variables *soc* and *supply* in the planetary lander case study, with continuous variable rounding and time discretisation to 0.1

8.5 THE BATCH CHEMICAL PLANT

The last case study is the *Batch Chemical Plant*, first presented by Kowalewski [113]. The goal is to produce saline solution at a given concentration. If part of the product is not used, the plant can recycle it to restart another production cycle.

This case study has been tackled in the VHS (Verification of Hybrid Systems) project [166] to (1) make a plan for the synthesis of saline solution and (2) to *verify* that the control routines for the single steps work correctly. In particular, our main contributions are in (1) formalisation, through PDDL+ language of the domain dynamics and (2) in the synthesis of the optimal universal plan for the system that, starting from a set of initial plant's configurations, produces saline solution recycling the unused part for the next production phase minimising the production time.

The plant (shown in Figure 8.13) is composed of 7 tanks connected through a complex pipeline, whose flow is regulated by 26 valves and two pumps. In particular, tank 5 is provided with a heater, whereas tank 6 is connected to a condenser. Finally, tanks 6 and 7 are surrounded by a cooling circuit. A set of sensors provide information to the plant controller about the filling level of tanks 1,2,3 and 5, the pump pressure and the condenser status.

In the plant initial state, all the valves are closed, and the pumps, heaters and coolers are switched off. Tank 1 contains saline solution at a high concentration c_{high}, whereas tank 2 contains water.

If tank 1 does not contain enough solution, the plant enters the *startup phase*: water from tank 2 is moved to tank 3, where a suitable amount of salt is added manually to reach the required concentration, and finally pumped to tank 1. Note that tank 2 can be refilled with water at any time by opening the appropriate input valve.

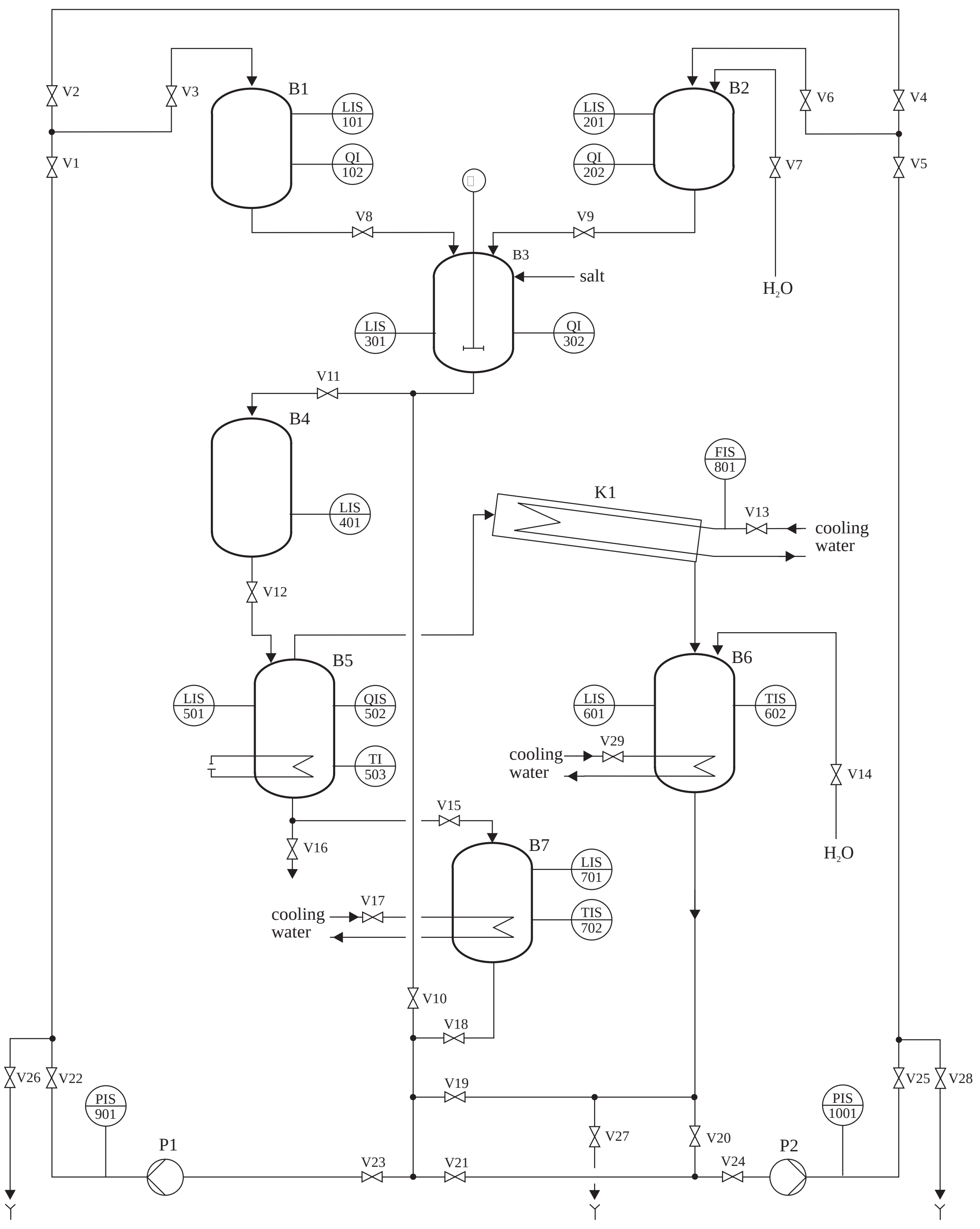


Figure 8.13: Overall structure of the batch chemical plant

When tanks 1 and 2 are appropriately filled, the plant can start the *production phase*. Tank 3 is partially filled with the solution from tank 1, which is then diluted using the water from tank 2 up to the requested concentration.

The resulting saline solution can be taken from the output valve of tank 3. If the product is not completely used, the plant recycles it in the next production cycle. To this aim, the solution in tank 3 is moved to tank 4 and then to tank 5. Here, the solution is boiled by the heater until it reaches the concentration c_{high}, and then moved to tank 7. The steam produced by this process is piped to the condenser that fills tank 6 with the resulting water. Finally, tanks 6 and 7 are cooled and their contents are pumped to tanks 2 and 1, respectively.

During the startup and production cycles the plant must obey some **safety constraints**:

1. pumps can be switched on only if all the valves in their pipeline are open,
2. the heater cannot be switched on if tank 5 is empty, or the condenser is switched off, or if the valves involved in the heating/condensation process are closed,
3. only two cooling circuits (including the one used by the condenser) can be switched on at the same time,
4. tanks cannot be filled and emptied at the same time,
5. the content of each tank must not exceed the corresponding capacity limitations [113], which are lower than the tank volume.

A_k	cross section of tank *k*
c_k	saline concentration in tank *k*
$c_{p,j}$	heat capacity of solution in tank *j*
$\Delta h_{vap,s}$	vaporisation enthalpy of solution *s*
h_k	filling level of tank *k*
$K_{k,l}$	volume flow from tank *k* to tank *l*
P_{heat}	heating power
$P_{cool,k}$	cooling power for tank *k*
ρ_j	density of solution in tank *j*
T_k	temperature of solution in tank *k*
$\dot{V}_i$	volume flow through valve *i*
$\dot{V}_{pk,l}$	volume flow through pump from tank *k* to tank *l*
$a_{k,l}$	section of pipe between tanks *k* and *l*
$H_{k,l}$	length of pipe between tanks *k* and *l*
$\zeta_{k,l}$	resistance of pipe between tanks *k* and *l*

Table 8.12: Batch chemical plant constants

8.5.1 Domain Specification

The plant dynamics is described by [57] through a set of differential equations. In particular, given the constants and variables shown in Table 8.12, the following equations describe the variation of the filling level for tanks directly connected by a pipe with an open valve (and possibly a pump switched on) during the startup phase:

$$
\begin{aligned}
A_2 \frac{dh_2}{dt} &= \dot{V}_7 && (8.4)\\
A_2 \frac{dh_2}{dt} &= -\dot{V}_9 = -K_{2,3}x; x \in [1; x_{2,max}] && (8.5)\\
A_3 \frac{dh_3}{dt} &= \dot{V}_9 = K_{2,3}x; x \in [1; x_{2,max}] && (8.6)\\
A_3 \frac{dh_3}{dt} &= -\dot{V}_{p3,1} && (8.7)\\
A_1 \frac{dh_1}{dt} &= \dot{V}_{p3,1} && (8.8)\\
& && (8.9)
\end{aligned}
$$

whereas the following equations describe the same variation for tanks involved in the production phase:

$$
\begin{aligned}
A_1 \frac{dh_1}{dt} &= -\dot{V}_8 = -K_{1,3}x; x \in [1; x_{1,max}] && (8.10)\\
A_3 \frac{dh_3}{dt} &= \dot{V}_8 = K_{1,3}x; x \in [1; x_{1,max}] && (8.11)\\
A_2 \frac{dh_2}{dt} &= -\dot{V}_9 = K_{2,3}x; x \in [1; x_{2,max}] && (8.12)\\
A_3 \frac{dh_3}{dt} &= \dot{V}_9 = K_{2,3}x; x \in [1; x_{2,max}] && (8.13)\\
A_3 \frac{dh_3}{dt} &= -\dot{V}_{11} = -K_{3,4}x; x \in [1; x_{3,max}] && (8.14)\\
A_4 \frac{dh_4}{dt} &= \dot{V}_{11} = K_{3,4}x; x \in [1; x_{3,max}] && (8.15)\\
A_4 \frac{dh_4}{dt} &= -\dot{V}_{12} = -K_{4,5}x; x \in [1; x_{4,max}] && (8.16)\\
A_5 \frac{dh_5}{dt} &= \dot{V}_{12} = K_{4,5}x; x \in [1; x_{4,max}] && (8.17)\\
A_5 \frac{dh_5}{dt} &= -\dot{V}_{12} = -K_{5,7}x; x \in [1; x_{5,max}] && (8.18)\\
A_7 \frac{dh_7}{dt} &= \dot{V}_{12} = K_{5,7}x; x \in [1; x_{5,max}] && (8.19)\\
& && (8.20)
\end{aligned}
$$

here, $K_{k,l} = \sqrt{\frac{2ga_{k,l}^2 H_{k,l}}{1+\zeta_{k,l}}}$ and $x = \sqrt{\frac{h_k}{H_{k,l}} + 1}$.

The variation of the filling level in tanks 5 and 6 is expressed differently, due to the effects of evaporation and condensation, respectively:

$$A_5 \frac{dh_5}{dt} = \frac{\dot{m}_{vap}}{-\rho_{sol}} \tag{8.21}$$

$$A_6 \frac{dh_6}{dt} = \frac{\dot{m}_{vap}}{\rho_w} \tag{8.22}$$

Equations are also given to calculate the variation of concentration and temperature in the tanks. The following equations compute the solution concentration in tanks 3 and 5:

$$A_3(c_3 \frac{dh_3}{dt} + h_3 \frac{dc_3}{dt}) = \dot{V}_9 c_2 \tag{8.23}$$

$$A_5(c_5 \frac{dh_5}{dt} + h_5 \frac{dc_5}{dt}) = -\dot{m}_{vap} c_5 \tag{8.24}$$

similarly, the temperature of tanks 5,6,7 is computed by the following equations:

$$c_{p,sol}\rho_{sol}A_5h_5 \frac{dh_{T_5}}{dt} = P_{el} \tag{8.25}$$

$$T_5 c_{p,sol}\rho_{sol}A_5 \frac{dh_{T_5}}{dt} = P_{el} - \dot{m}_{vap}(c_{p,sol}T_5 + \Delta h_{vap}) \tag{8.26}$$

$$c_{p,lo}\rho_{lo}A_7h_7 \frac{dT_7}{dt} = -P_{cool} \tag{8.27}$$

$$c_{p,w}\rho_w A_6 h_6 \frac{dT_6}{dt} = -P_{cool} \tag{8.28}$$

8.5.2 System Modelling

The most challenging and interesting aspect of the chemical plant specification is the production phase, so in the following we will focus only on the modelling of this phase.

This continuous, time-dependant domain is mainly modelled using processes, events and (flexible) durative actions. Indeed, Figures 8.14, 8.16 and 8.17 show representative examples of such constructs extracted from the model (whose full source is available online in [47]), which contains a total of 59 predicates, 55 functions (14 of which represent real values), 19 events, 10 durative actions and 11 processes. In the figures, `B`*x*`_l`, `B`*x*`_c`, `B`*x*`_t` indicate the filling level, solution concentration and temperature for tank *x*, respectively, whereas `V`*y*, `P`*y* and `H`*y* indicate valve, pump and heater *y*, respectively. Finally, the value of a constant *k* taken from the problem specification is indicated with `c_`*k*.

In the following we describe the main elements of the PDDL+ model for the chemical plant production phase, highlighting their most interesting features. It is worth noting that the model has been written to adhere as much as possible to the formal specification given by [57]. However, to further check its correctness, we extracted from the universal plan generated by UPMurphi the single production policy corresponding to the initial conditions described by [113] and we verified that it was identical to the one (manually) devised by [113].

```
; filling durative action (for tank 3)
(:durative-action B3_fill
:parameters ()
:duration (>= ?duration 0)
:condition (and
 (at start (not (V8))) (at start (= (B3_l) 0))
 (at start (>= (B1_l) 0)) (at start (not (V3)))
 (at start (not (V10))) (at start (not (V11)))
 (at start (not (B3_filled))) (at end (V8))
 (over all (>=(B1_l) 0)))
:effect (and
 (at start (B3_filling)) (at start (V8))
 (at end (not (V8))) (at end (B3_filled))
 (at end (not (B3_filling)))))
; filling process (for tank 3)
(:process B3_fill_process
:parameters ()
:precondition (B3_filling)
:effect (and
 (decrease (B1_l) (* #t (* (C_5_2) (sqrt (+ (/ (B1_l) (C_h_1_3)) 1 )))))
 (increase (B3_l) (* #t (* (C_5_2) (sqrt (+ (/ (B1_l) (C_h_1_3)) 1 )))))))
```

Figure 8.14: Examples of durative actions and processes modelling the production phase of the batch chemical plant

Production Activities. The production activities, such as moving the solution from a tank to another, cool it down, etc., some of which can possibly be executed in parallel, are modelled using durative actions. However, the duration of these activities is not known *a priori*, thus the planner should determine the time point at which the tank capacity (or required concentration, or temperature) is reached. To achieve this, we use *duration inequalities* in the durative actions. On the other hand, continuous change to solution level, concentration and temperature in tanks are modelled through PDDL+ processes that update the corresponding model variables following the functions described by [57]. This modelling schema guarantees an immediate detection (i.e., triggering of failure events) of safety violations.

As an example, when tank 1 is nonempty, tank 3 is empty and some other conditions hold, the durative action PDDL B3_fill shown in Figure 8.14 moves the solution from tank 1 to tank 3. The continuous update to the solution level in these tanks due to the durative action is performed by the process PDDL B3_fill_process, which is enabled by the durative action by setting to true the predicate PDDL B3_filling. The execution of this process may in turn trigger some events [76], e.g., PDDL B3_l_failure (shown in Figure 8.16) that would invalidate the plan. At the end of the durative action (as chosen by the planner), PDDL B3_filling is set to false, and the filling process ends.

```
(:process B3_fill_process
:parameters ()
:precondition (B3_filling)
:effect (and
 (decrease (B1_l)
   (* #t (* (C_5_2)( + (* -0.000415797 (* (B1_l) (B1_l) ) ) (+ (* (B1_l)
   0.0424115 ) 1.00597 )))))
 (increase (B3_l)
   (* #t (* (C_5_2)( + (* -0.000415797 (* (B1_l) (B1_l) ) ) (+ (* (B1_l)
   0.0424115 ) 1.00597 ))))))
)
```

Figure 8.15: PDDL B3_fill_process with approximated square root

It is worth noting that the effects of PDDL B3_fill_process involve the calculation of a square root, which is currently not supported by PDDL+. Therefore, we have also created and tested an *approximated* model (available in [47]), where the square root is substituted by the second degree polynomial on the variable $B1\_l$ that best fits such function within the bounds deducible from the model dynamics. The corresponding approximated PDDL B3_fill_process is shown in Figure 8.15.

```
; pipeline flow failure (during B3 filling process)
(:event B3_flow_failure
:parameters ()
:precondition (and (or (V11) (V10)) (or (V8) (V9)))
:effect (not (correct_operation)))
; heater failure (on tank 5)
(:event H5_failure
:parameters ()
:precondition (or
  (and (H5) ( or (V12) (V15) (V16 )))
  (and (H5) (not(V13)))
  (and (H5) (not (>= (B5_l) (B5_l_safe)))))
:effect (not (correct_operation)))
; tank filling limit failure (on tank 3)
(:event B3_l_failure
:parameters ()
:precondition (or (< (B3_l) 0) (> (B3_l) (B3_l_max)))
:effect (not (correct_operation)))
; pump (2) failure
(:event P2_failure
:parameters ()
:precondition (and (P2) (not(or
    (and (V25) (V28))
    (and (V25) (V5) (V6))
    (and (V25) (V5) (V4) (V2) (V1) (V3)))))
:effect (not (correct_operation)))
```

Figure 8.16: Examples of failure events of the batch chemical plant

Production Events. The violation of one of the safety constraints should trigger an instantaneous change that invalidates the plan. Therefore, such failures have been modelled through PDDL+ events, whose effect is to falsify the invariant predicate PDDL correct_operation.

It is worth noting that, in the chemical plant model, discrete and continuous change are combined in the activation conditions of several events [97], making their checking more complex, but still very important since they may invalidate the plan [74]. As an example, event PDDL H5_failure in Figure 8.16 shows the PDDL+ model of an exogenous event. Such event is activated when the heater is switched on (PDDL H5 is true) and one of the valves 12, 15 or 16 is open (PDDL or V12 V15 V16), or valve 13 is closed (PDDL not V3), or the level of tank 5 is lower than the security level (PDDL not (>= B5_l B5_l_safe)).

Finally, the two events shown in Figure 8.17 are used to trigger the end of the plan. In particular, event PDDL production_end is triggered when tank 1 contains a sufficient amount of solution with the required concentration, and its effect is to set the PDDL production_complete predicate to true. This, in turn, triggers a *cascading* event PDDL production_success that, if the plant has operated correctly (i.e., without violating any safety constraint) and all the valves and pumps have been correctly closed, sets the PDDL success predicate to true to indicate that the goal has been reached.

```
(:event production_end
:parameters ()
:precondition (and
 (B1_filled) (>= (B1_l) (B1_l_target_min))
 (< (B1_l) (B1_l_target_max))
 (= (B1_c) (B1_c_target)) (not(production_ended)))
:effect (and (production_complete)
 (production_ended)))
(:event production_success
:parameters ()
:precondition (and (not(success))
 (production_complete) (correct_operation)
 (not (or (V1) (V2) (V3) (V4) (V5) (V6) (V7) (V8) (V9) (V10) (V11) (V12) (V13) (
  V14) (V15) (V16) (V17) (V18) (V19) (V20) (V21) (V22) (V23) (V24) (V25) (V26) (
  V27) (V28) (V29) (P1) (P2))))
:effect (success))
```

Figure 8.17: Cascading events triggering the goal of the batch chemical plant

Production Problem. The PDDL+ definition of the problem for the batch chemical plant production phase is quite straightforward. The domain is initialised by setting the function and predicate values to the ones obtained after the startup phase (see [57]), and the goal is to set the PDDL success predicate to true, minimising the PDDL total-time.

8.5.3 PDDL+ MODEL

We want to use UPMurphi to automatically perform universal planning on the startup and production phases, in order to generate a *set of policies* for the system.

To this aim, we fist discretise the PDDL+ model, as suggested by [28], by rounding up the continuous variables up to the first decimal, and the time in steps of 10 seconds. Then, to generate the *start state clouds* used to initialise our universal planning engine, we proceed as follows.

The startup phase is triggered before a new production cycle if tank PDDL B1 is empty or does not contain enough saline solution at concentration c_{high} (possibly recycled from the previous production phase). In this case, the startup phase must fill PDDL B1 up to PDDL B1_l_max. Thus, the *start state cloud* for this phase considers all the values for PDDL B1_l in the range $[0,$ PDDL B1_l_max$]$ with PDDL B1_l_max = 8 liters (as specified in [113]) and steps of 0.1 liters, i.e., 81 different start states.

On the other hand, the production phase, thanks to the startup postconditions, always starts working on a plant where PDDL B1 and PDDL B2 are completely filled. Here, the only parameter used to define the start state cloud is the amount of solution to be produced, that is PDDL B3_l_target. We vary this value in the range $[1.5,\ldots,3.7]$ liters with steps of 0.1, obtaining 23 different start states.

8.5.4 Universal Planning

Figure 8.13 shows the generation statistics for the startup phase universal plan. The plant state space is 10^{17}, however, starting from the given start state cloud, the planner found that only about 3 million of such states were actually reachable, and only for 22% of them is was possible to calculate a (optimal) policy to reach the goal.

State space size	10^{29}
Start state cloud size	81
Reachable states	$3,092,112$
States to goal (generated plans)	$679,193$
Synthesis time	530 sec
Peak of memory required	61 MB

Table 8.13: Batch chemical plant startup phase universal plan generation statistics

On the other hand, the whole universal plan for the more complex production phase took about 6000 seconds to be generated, as shown in Table 8.14. Indeed, in this case there were about 30 million of reachable states (which are still sensibly less than the state space size), and for 24% of them UPMurphi was able to generate an optimal plan to the goal.

State space size	10^{29}
Start state cloud Size	23
Reachable states	$29,968,861$
States to goal (generated plans)	$7,154,464$
Synthesis time	$6,319.8$ sec
Peak of memory required	630 MB

Table 8.14: Batch chemical plant production phase universal plan generation statistics

```
0.0: (B3_fill) [250]
260.0: (B3_dilution) [130]
400.0: (B4_fill) [290]
700.0: (B5_fill) [180]
890.0: (B5_evaporate) [750]
1650.0: (B7_fill) [130]
1790.0: (B7_cool) [270]
1800.0: (B6_cool) [160]
1970: (B2_fill) [120]
2070: (B1_fill) [80]
```

Figure 8.18: A planned production policy for the batch chemical plant

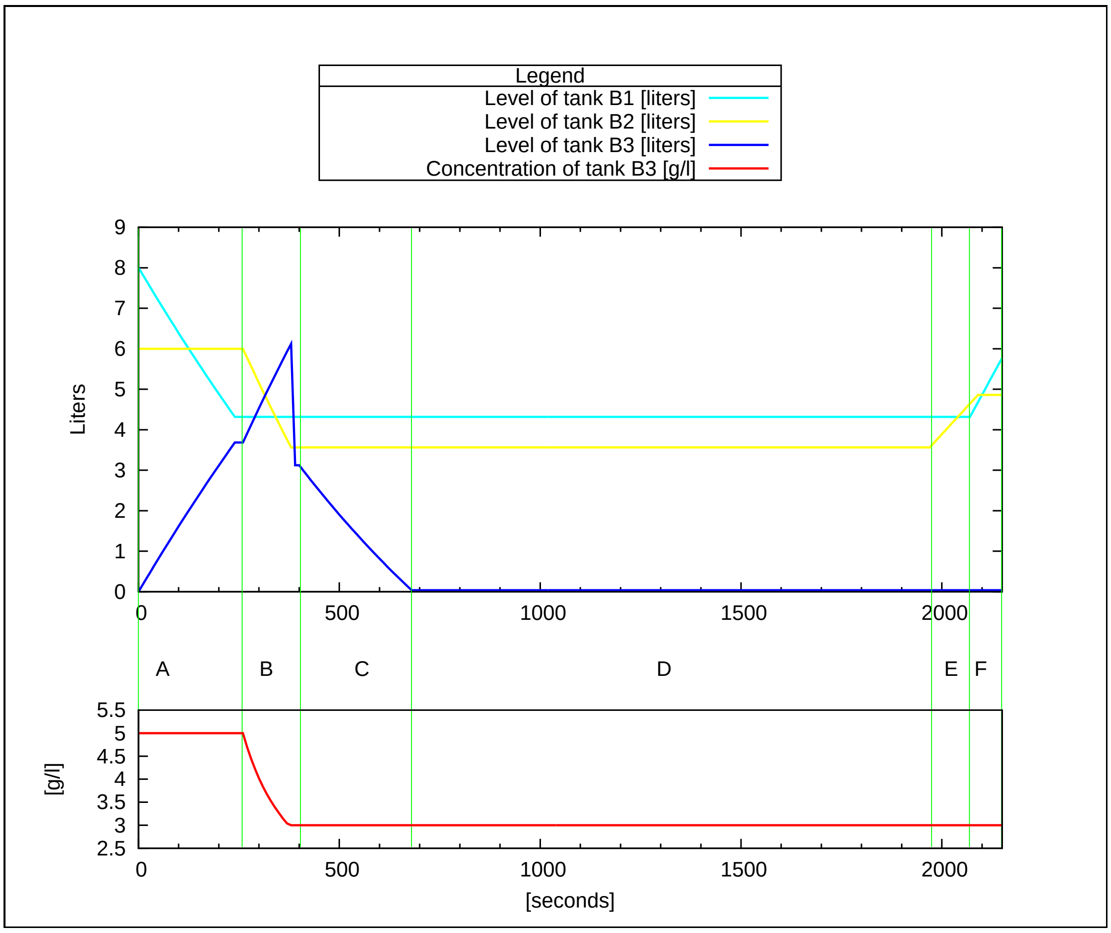


Figure 8.19: Variation of filling levels computed by VAL for tanks PDDL B1,PDDL B2, PDDL B3 and solution concentration in tank PDDL B3 during the batch chemical plant production cycle described in Figure 8.18

The validation of the generated plans confirmed that the initial discretisation was fine enough to obtain correct results.

As an example, Figure 8.18 shows one of the generated production policies, where PDDL B3_level_target = 3 liters. Figure 8.19 graphically shows the variation of PDDL B1_l, PDDL B2_l, PDDL B3_l and PDDL B3_c, respectively, as calculated by VAL during the validation of this plan. In particular, in the figure, letters A-F are used to indicate the time spans where the plant is performing particular tasks, i.e., A,B,C correspond to the activation of PDDL B3_fill, PDDL B3_dilution and PDDL B4_fill, respectively, D indicates the recycle phase, and E,F the activation of PDDL B2_fill and PDDL B1_fill, respectively.

We see that the filling level of the first two tanks initially decreases due to the execution of the PDDL B3_fill (span A) and PDDL B3_dilution (span B) processes, respectively, while PDDL B3 gets filled. On the other hand, the concentration PDDL B3_c remains stable on c_{max} during PDDL B3_fill, and rapidly decreases during the dilution (PDDL B3_dilution) up to c_{target}. Finally, part of the product is manually drained from PDDL B3, and the remaining solution is moved to other tanks (i.e., PDDL B3_l reaches zero, span C), where it is recycled (span D) and finally pumped back to PDDL B1 (span E) and PDDL B2 (span F).

CHAPTER 9

DATABASE DATA QUALITY ANALYSIS VIA MODEL CHECKING

9.1 MOTIVATION AND CONTRIBUTION

In the previous sections we discussed the problem of planning and control for continuous systems via model checking techniques. We showed how explicit model checking can be used to solve both planning and control problems for systems modelled on FSS.
In this section we extend the use of model checking to a class of problems far from planning and control: the *data quality* problem. Informally speaking, data quality is a general concept and it can be described by many dimensions, e.g., *accuracy, consistency, accessibility* (a complete survey on data quality dimensions is in [13]).
We focus on consistency, which is a dimension of data describing the violation of semantic rules defined over a set of data items, where items can be tuples of relational databases. Here we are interested in the evaluation of *consistency* by using model checking in search of inconsistencies on data sources, typically represented on database structure.

To this regard, we intend (1) to map a data quality problem on a FSS and (2) to use model checking to verify if the system holds them. Indeed, our idea is that formal methods, and model checking in particular, can be helpful in some specific data quality scenarios to automate data consistency verification, to make more robust the overall data quality process, and to improve domain understanding, since formal methods can facilitate knowledge sharing between technicians and domain experts. It is worth noting that evaluating cleansed data accuracy against real data is often either unfeasible or very expensive (e.g. lack of alternative data sources, cost for collecting the real data), then consistency based methods may contribute reducing the accuracy evaluation efforts.

To this regard, in this section we provide the following contributions:

- The definition of a methodology, namely the *Robust Data Quality Analysis* (RDQA), which uses formal methods to formalise consistency rules.
- The automatic verification of consistency rules on big datasets through model checking techniques (namely, the CMurphi model checker in this instance).

- The RDQA has been successfully exploited on a real industrial data quality case study of a Public Administration database provided by the C.R.I.S.P. research center [43].

A preliminary version of this work has been published in [131].

9.2 Introduction and Related Work

Our society is actually dependent from digital data, which now plays a crucial role in the Information and Communication Technology. One need only consider that business and governmental applications, web applications as well as relations between citizens and public administration is now based on electronic data. Hence, it is clear that the *quality* of digital data and the effects on every kind of analysis and information obtained from such data are crucial. To give an example, the causes of the Challenger Space shuttle explosion are imputed to ten different categories of data quality problems (see [72] for details). On the other hands, several studies (e.g. [160, 146, 13]) report that enterprise databases and Public Administration archives suffer from poor data quality, therefore before decisions makers may successfully exploit those data, their quality has to be accurately before the decision making processes.

Despite a lot of research effort has been spent and many techniques and tools for improving data quality are available, their application to real-life problems is still a challenging issue [123]. When alternative and trusted data sources are not available, the only solution is to implement cleansing activities relying on business rules, but it is a very complex, resource consuming, and error prone task.

Developing cleansing procedures requires strong domain and ICT knowledge. Diverse actor types are required (e.g. ICT and Business) who should collaborate, but knowledge sharing is hindered by their different cultural backgrounds and interpretation frameworks. Fort this reason, several cleansing tools have been introduced into the market, focusing on user friendly interfaces to make them usable by a broad audience.

From the research perspective, data quality has been addressed in different contexts, including statistics, management and computer science [148].

Our work focuses on improving database instance-level consistency. In such a context, research has mostly focused on *business rules*, *error correction* (known as both *data edits* and *data imputation* in statistics [148]), *record linkage* (known as *object identification*, *record matching*, and *merge-purge problem*), and *profiling* [71]. A description of several data cleansing tools can be found in [123, 11, 13, 135].

Even the adoption of cleansing techniques based on *statistical algorithms* or on *machine learning* requires a huge human intervention for assessment activities. Errors that involve relationships between one or more fields are often very difficult to uncover with existing methods. These types of errors require deeper inspection and analysis [123].

Similar considerations can be applied to *data profiling* tools. Data profiling is a blurred expression that can refer to a set of activities including data base and data warehouse reverse engineering, data quality assessment, and data issues identification.

According to [70] the principal barrier to more generic solutions to information quality is the difficulty of defining what is meant by high or poor quality in real domains, in a sufficiently precise form that it can be assessed in an efficient manner. This part of the thesis contributes to address the just described issue.

Many cleansing tools and database systems exploit *integrity analysis* (including *relational integrity*) to identify errors. While data integrity analysis can uncover a number of possible errors in a data set, it does not address complex errors [123]. Some research activities (e.g. [71]) focus on expanding integrity constraints paradigms to deal with a broader set of errors. In this streamline the approach we adopt contributes to manage a broader set of consistency errors with respect to the integrity constraints tools and techniques currently available.

The application of *automata theory* for inference purposes was deeply investigated in [165, 110] for the database domain. The approach presented in [3] deals with the problem of checking (and repairing) several integrity constraint types. Unfortunately most of the approaches adopted can lead to hard computational problems.

Only in the last decade formal verification techniques were applied to databases, e.g. *model checking* was used in the context of *database verification* [40] to formally prove the termination of triggers. Model checking has been used to perform data retrieval and, more recently, the same authors extend their technique to deal with CTL in order to solve queries on semistructured data [61].

In the end, to the best of our knowledge no contribution in literature has exploited formal methods for analysing the quality of (real-life) database contents. Indeed formal methods contribute to manage a broader set of consistency errors with respect to the integrity constraints tools and techniques currently available.

9.3 Finite State Events Database

Several database contents can be modelled as sequences of events (and related parameters), where the possible event types being a finite set. For example, the registry of (university) students' scores, civil registries, the retirement contribution registry, several public administration archives, and financial transaction records may be classified in such category. The event sequences that populate such databases might be modelled by FSS (according to Definition 2), which in turn open several possibilities with respect to *consistency check* and *data quality* improvement. FSS can be used to model the domain business rules so that the latter can be automatically checked against database contents by making use of formal methods, e.g. Model Checking. Furthermore FSS representations can be easily understood by domain experts and by ICT actors involved in Data Quality improvement activities.

Our approach is based on the idea that Model Checking tools can be used to evaluate the consistency of databases both before and after the application of data cleansing activities. By comparing the consistency check results of the two database instances (before and after the cleansing process), it is possible to obtain useful insight about the implementation of the cleansing procedures. This evaluation helps improving the data cleansing development processes since feedbacks can be achieved on the consistency of the results.

Developing a cleansing procedure for a large domain may be a very complex task which may require to state several business rules, furthermore their maintenance could be an onerous task, since the introduction of new rules may invalidate some of the existing ones. The possibility to model a correct behaviour using FSS formalisms and to check the results of data cleansing can effectively reduce the effort of designing and maintaining cleansing procedures. Then, we define “Finite State Event Dataset” (FSED) and “Finite State Event Database” (FSEDB) as follows.

Definition 15 (Finite State Event Dataset)**.** *Let $\varepsilon = e_1, \ldots, e_n$ be a finite sequence of events, we define a* Finite State Event Dataset *(FSED) as a dataset S whose content is as a sequence of events $S = \{\varepsilon\}$ that can be modelled by a Finite State System.*

Definition 16 (Finite State Event Database)**.** *Let S_i be a FSED, we define a* Finite State Event Database *(FSEDB) as a database DB whose content is $DB = \bigcup_{i=1}^{k} S_i$ where $k \geq 1$.*

We introduced the set of sequences in the FSEDB definition since many database contents can be easily modelled by splitting their content is several subsets (each being a sequence of events) and then modelling each sequence with a single (or a limited set of) FSS. Although the whole content could be modelled by a single FSS, splitting into subsets can reduce the complexity of the FSS(s) used to model the sequences. Many Public Administration archives can be classified as Finite State Event Databases, and the possibility to use FSS formalisms to improve cleansing activities is extremely valuable.

9.4 Robust Data Quality Analysis

In the following, we describe our *Robust Data Quality Analysis* (RDQA). Roughly speaking, assume clr to be a function able to clean a source (and dirty) dataset into a cleansed one according to some defined cleansing rules (or *business rules*). To this regard, we can take on loan the definition given in [12] where consistency refers to *“the violation of semantic rules defined over a set of data items. With reference to the relational theory, integrity constraints are a type of such semantic rules. In the statistical field, data edits are* *typical semantic rules that allow for consistency checks"*. In this settings, several questions arise: *"what is the degree of consistency achieved through clr? Can we improve the consistency of the cleansed dataset? Can we be sure that function clr does not introduce any error in the cleansed dataset?"* .

The set DB_S represents a dirty database whilst DB_C is the cleansed instance of DB_S computed by function clr working iteratively on each subset $S_i \subseteq DB_S$ where $C_i = clr(S_i)$ and $C_i \subseteq DB_C$. Since many consistency properties are defined or scoped on portions of the original database, the cleansing activity is not carried out on the whole dataset DB_S but on several subsets S_i of the original one.

The clr function applied to S_i may produce: a C_i that is *unchanged* with respect to S_i (in case S_i had a good quality); or it may produce a *changed* C_i (in case some quality actions have been triggered). Since the semantics of the *changed/unchanged* are domain dependent, an *equals* function which looks for equality between S_i and C_i is required.

Moreover, since the function *clr* might not effectively cleanse the data, an evaluation of its behaviour is carried out using a further function *ccheck* which is based on formal methods. *ccheck* is used to verify the consistency of both S_i and C_i. Several outcomes of the cleansing routines can be identified in this way e.g., a dirty S_i may have been cleansed into a consistent C_i, or a dirty S_i may have been turned into a not consistent C_i, or a clean S_i may have been modified into a not consistent C_i.

Nevertheless, even if *ccheck* is based on formal methods, no enough guarantees are given about the correctness of *ccheck* (i.e., we cannot use *ccheck* as an oracle). Instead, the compared results given by functions $ccheck, equals$, and clr allow one to obtain useful insights about the consistency of the clr function and at the same time it is helpful to evaluate the *ccheck* and *equals* functions. This procedure will be further detailed in the following paragraph by means of examples. For the sake of clarity, we formally describe the RDQA process defining the following functions:

Function 1 (clr)**.** *Let S be a dataset according to Definition 15, then* $clr : S \rightarrow C$ *is a total function where C represents the cleaned instance of S.*

Function 2 (rep)**.** *Let X be a dataset according to Definition 15, then* $rep : X \rightarrow e$ *is a total function which returns a representative element* $e \in X$.

Function 3 (*ccheck*)**.** *Let K be a dataset according to Definition 15, then* $ccheck : K \rightarrow \{0,1\}$ *where* $ccheck(K)$ *returns* 1 *if exists a sequence* $\varepsilon \in K$ *such that* ε *contains an error,* 0 *otherwise.*

Clearly, function *ccheck* can be realised by using any formal method. In such context, we use Model Checking techniques. In this case, we use the CMurphi model checker, as described in Section 9.5.4.

Function 4 ($equals$)**.** *Let S and C be datasets according to Definition 15 we define* $equals : S \times C \rightarrow \{0,1\}$ *which returns* 0 *if no differences between S and C are found,* 1 *otherwise.*

The RDQA procedure is applied iteratively refining at each step the functions *clr* and *ccheck* until a desired consistency level is reached. In Fig. 9.1 it is shown a graphical representation of a RDQA iteration whilst Tab. 9.1b outlines the semantics of the F_S^{+-}, F_C^{+-}, and D^{+-} sets, which are used in Tab. 9.1a and Fig. 9.1. Each iteration computes the *Double Check Matrix* (DCM), e.g. Tab. 9.1a, where the just introduced information are summarised in order to analyse the reached consistency level. For sake of completeness, a pseudo-code of the RDQA approach is given in Procedures 9,10 and 11. The function *compute_DCM* is implemented as MySQL stored procedures whilst function *ccheck* is realised using the CMurphi model checker.

To give some example of the information provided by the DCM of Figure 9.1a, row 1 gives the number of items for which no error was found by *ccheck* applied both on S_i and C_i, and no differences between the original instance and the cleansed one was found by *equals*. In this case both *ccheck* and *clr* agreed that the original data was clean and no intervention was needed. Differently, row 4 shows the number of items for which no error was found by $ccheck(S_i)$ whilst the $equals(S_i, C_i)$ states that a cleansing intervention took place, producing a wrong results recognised as dirty by $ccheck(C_i) = 1$. The case identified by row 4 is very important since it discloses bugs either in in the cleansing procedure, or in the *clr*, or in the *ccheck* function (or a combination thereof). Row 8 shows another interesting case, where it is reported the number of items that where originally dirty ($check(S_i) = 1$), an intervention took place ($equals(S_i, C_i) = 1$) that was not effective since $ccheck(C_i) = 1$. The other cases will be extensively commented in Section 9.5 on a real example.

Is worth to note that, thanks to the comparisons outlined, the DCM can be used as a *bug hunter* to start an improvement process which lead to better understand the domain rules, and to refine the implementation of the cleansing activities. The RDQA approach does not guarantee the correctness of the data cleansing process, nevertheless it helps making the process more robust with respect to data consistency.

Procedure 9 RDQA

```
S =get_source_dataset();
$D^+ = \emptyset; D^- = \emptyset$;
$F_S^+ = \emptyset; F_S^- = \emptyset$;
$F_C^+ = \emptyset; F_C^- = \emptyset$;
for all $S_i \subseteq S$ do
  $C_i = clr(S_i)$;
  compute_$equals(S_i, C_i)$;
  compute_$ccheck(S_i)$;
  compute_$ccheck(C_i)$;
end for
compute_$DCM()$; // As shown in Table 9.1a
display_DCM();
```

Table 9.1: (a) The Double Check Matrix. (b) The definition of sets resulting by *ccheck* and *equals* functions

(a)

Conditions			*Result*
$ccheck(S_i)$	$equals(S_i,C_i)$	$ccheck(C_i)$	Cardinality
0	0	0	$\|F_S^- \cap D^- \cap F_C^-\|$
0	0	1	$\|F_S^- \cap D^- \cap F_C^+\|$
0	1	0	$\|F_S^- \cap D^+ \cap F_C^-\|$
0	1	1	$\|F_S^- \cap D^+ \cap F_C^+\|$
1	0	0	$\|F_S^+ \cap D^- \cap F_C^-\|$
1	0	1	$\|F_S^+ \cap D^- \cap F_C^+\|$
1	1	0	$\|F_S^+ \cap D^+ \cap F_C^-\|$
1	1	1	$\|F_S^+ \cap D^+ \cap F_C^+\|$

(b)

$F_S^- = \bigcup(rep(S_i) \| ccheck(S_i) = 1)$
$F_S^+ = \bigcup(rep(S_i) \| ccheck(S_i) = 0)$
$F_C^+ = \bigcup(rep(C_i) \| ccheck(C_i) = 0)$
$F_C^- = \bigcup(rep(C_i) \| ccheck(C_i) = 1)$
$D^- = \bigcup(rep(S_i) \| equals(S_i,C_i) = 0)$
$D^+ = \bigcup(rep(S_i) \| equals(S_i,C_i) = 1)$

Procedure 10 COMPUTE_EQUALS

Input: S_i, C_i

```
1: if $(equals(S_i,C_i) = 1)$ then
2:   $D^+ = D^+ \cup rep(S_i)$;
3: else
4:   $D^- = D^- \cup rep(S_i)$;
5: end if
```

Procedure 11 COMPUTE_CCHECK

Input: X_i // It can be S_i or C_i

```
1: if $(ccheck(X_i) = 1)$ then
2:   $F_X^+ = F_X^+ \cup rep(X_i)$;
3: else
4:   $F_X^- = F_X^- \cup rep(X_i)$;
5: end if
```

9.5 An Industrial Application: The Worker Career Administrative Archive

The RDQA approach has been tested on a real case scenario. The C.R.I.S.P. research center [43] exploits the content of a Public Administration database to study the labour market dynamics at territorial level [124]. A lot of errors and inconsistencies (missing information, incorrect data, etc.) have been detected in the database, therefore a cleansing process is executed, and the RDQA approach has been used to improve such process.

9.5.1 Domain Description

According to the Italian law, every time an employer hires or dismisses an employee, or a contract of employment is modified (e.g. from part-time to full-time, or from fixed-term contract to unlimited-term) a communication (Mandatory Communication hereafter) is sent to a registry (job registry hereafter) by the employer. The registry is managed at *provincial level*, so every Italian province has its own job registry recording the working history of its inhabitants. An Italian province is an administrative division which encompass a set of cities and towns geographically close. In this scenario, the database of a province is used by the C.R.I.S.P. to extract longitudinal data upon which further analysis are carried out.

Every mandatory notification (event hereafter) contains several data, among which the most important for the purpose of our work are: event_id (a numeric id identifying the communication), employee_id (an id identifying the employee), event_data (the communication date), event_type (whether it is the start, the cessation, the extension or the conversion of a working contract), full_time_flag (a flag stating whether the event is related to a full-time or a part-time contract), employer_id (an identifier of the employer), contract_type (e.g. fixed-term contract, unlimited-term contract, Apprenticeship, etc.).

9.5.2 Career (simplified) Model

For the sake of simplicity, we have modelled a set of events which maps onto the mandatory communications data. The events are:

Start: the worker has signed a contract and has started working for an employer. Further information describing the event are: the date, the employee_id, the employer_id, the contract_type, the full_time_flag.

Cessation: the worker has stopped working and the contract is terminated. Further information describing the event are: the date, the employee_id, the employer_id.

Extension: a fixed-term contract has been extended to a new date. Further information describing the event are: the current date, the employee_id, the employer_id, the new termination date.

Conversion: a contract type has changed, e.g. from fixed-term to unlimited-term contract. Further information describing the event are: the date, the employee_id, the employer_id, the new contract_type.

Some business rules can be inferred by the Italian Labour Law which states for examples that an employee can have only a full-time contract active at the same time, or alternatively no more than two part-time contracts. According to the law, the career of a person showing two start events of a full-time contract without a cessation in between is to be considered invalid. Such errors may happen when Mandatory Notifications are not recorded or are recorded twice. In our context a job career is a temporal sequence of events describing the evolution of a worker's state, starting from the beginning of her/his working history. It is worth noting that a person can have two contracts at the same time only if they are part-time and they have been signed with two different organisations.

9.5.3 Graph Representation

Figure 9.2 shows a simplified representation of the evolution of a job career where nodes represent the state of a worker at a given time (i.e., the number of active part-time/full-time contracts) whilst edges model how an event can modify a state.

To give an example, a valid career can evolve signing two distinct part-time contracts, then proceeds closing one of them and then converts the last part-time into a full-time contracts (i.e., $unemp, emp_1$, emp_2,emp_1, emp_4). For sake of clarity, Figure 9.2 focuses only on nodes/edges describing a correct evolution of a career (i.e., the white nodes) whilst all other nodes/edges are omitted (e.g., careers having events related to unsubscribed contracts). Nevertheless, Figure 9.2 contains two filled nodes (i.e. emp_5, emp_6) which are helpful to describe some invalid careers. To this regard, a career can get wrong subscribing three or more part-time contracts (i.e., $unemp, emp_1$, emp_2, emp_6) or activating both part-time and full-time contracts (i.e., $unemp, emp_1$, emp_2, emp_5).

9.5.4 The CMurphi Model

In this section we closely look at the realisation of the *ccheck* function according to the definition of Function 3. It is clear that our scenario represents a FSEDB (as in Definition 16), thus it can be modelled as a finite state system, applying model checking technique to *verify* each FSED (i.e., each worker's career). To this aim, we use the CMurphi model checker (introduces in Section 3.4) to realise a procedure able to analyse the

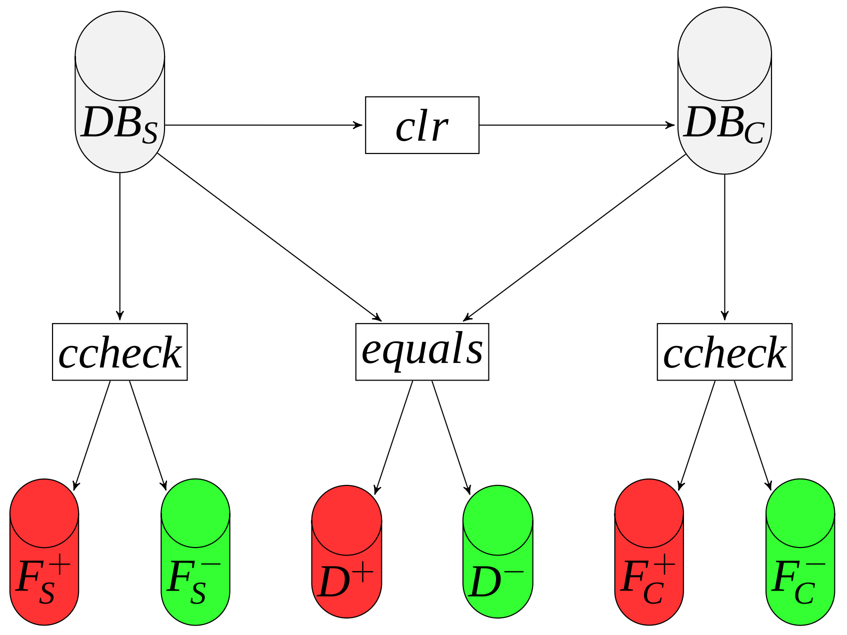


Figure 9.1: A schematic view of the Robust Data Quality Analysis iteration.

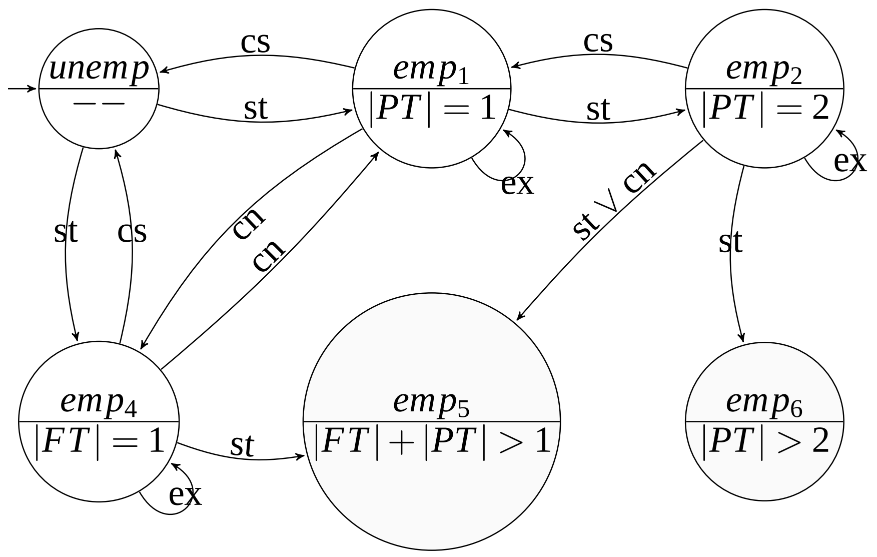


Figure 9.2: An Abstract representation of the dynamics of a job career where $st = start$, $cs = cessation$, $cn = conversion$, and $ex = extension$.

system domain and verify its correctness with respect to a given semantics (i.e., the Italian low).

Figure 9.3 shows the overall structure of the *ccheck* implementation, which works as follows:

CMurphi Model is shown in Figure 9.4. In the first part declarative statements (which are used to model the employment state of a worker), declarations of constants, datatypes, and external C/C++ functions are declared.

The dynamics of the system is composed by a startstate (which models the *unemployed* state of a worker), and by a transition rule `next_event` to compute the evolution of a career reading events directly from the corresponding dataset. The `ruleset` keyword is used make parametric the system dynamics (i.e., the range `min_worker..max_worker` represents a start state cloud to adapt the number of worker to verify in a single validation process).

Thanks to this, CMurphi automatically generates all possible startstates within the given range, starting a verification for each state reachable from the initial ones and returning one (or more) error trace (i.e., invalid careers) for each worker, making the model more scalable in case of huge databases.

Finally, the last part of Figure 9.4 models properties which must be satisfied along a career, defined by using a C++ external function `safe_transition`, by verifying that *each* `invariant` clause is always satisfied along any career evolution and returning the error trace (i.e., set of careers violating at least an invariant clause). Note that the output can be easily used for the RDQA methodology, as described in Section 9.4. In refer to example of Figure 9.2, the trajectory $(< unemp, st >< emp_1, st > < emp_2, cn >< emp_5 >)$ violates both the `Part-time` and `Full-time` invariants of Figure 9.4.

CMurphi Engine is the verification algorithm of CMurphi, as presented in Section 3.4. It is worth noting that we modified it to obtain (1) exactly one error trace (if any) for each career and (2) to allows CMurphi to store the error trace on database (i.e., CMurphi directly writes databases $F_S^+, F_S^-, F_C^+, F_C^-$).

DBMS is used to allows the interaction between *CMurphi Engine* and the database. To this regard, in order to speed up the queries on database, C++ external functions play an important role to retrieve data from database and to store it on a C++ buffer, easily accessible to CMurphi via external functions. The DBMS is a MySQL 5.4 server whilst the connection interface between MySQL and C++ are realised using the `cppconn` connector provided by the Boost library [163].

Finally, an example of CMurphi validation on a set of careers is given in Figure 9.5 (we modified CMurphi to return errors both on file and on database). In particular, worker number 73 starts and closes a part-time contract with company 23680 and a full-time contract with company 9165. Then, it reopen a part-time contract with the same company,

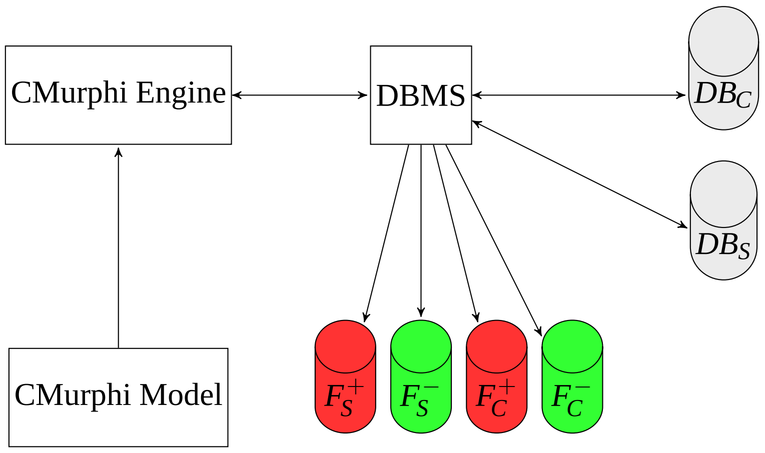


Figure 9.3: Representation of the *ccheck* implementation using the CMurphi model checker.

which is closed later as a full-time contract. In this case, invariant *valid cessation event* is violated.

9.5.5 Robust Data Analysis: Experimental Results

In this section we show some experimental results performed on an administrative database DB_S having $1,248,752$ events (i.e. $|DB_S|$) and $213,566$ careers (i.e., all distinct subsets S_i where $i \in [1,\ldots,213566]$). Note that the results are referred to the first iteration of the RDQA process described in Section 9.4 in order to highlight how the RDQA process was useful to identify inconsistencies in real data cleansing operations.

The application of the function *clr* (defined according to Function 1) on DB_S generated a new dataset DB_C with $|DB_C| = 1,089,895$. Then, the function *ccheck* has been realised according to Definition 3, using the CMurphi model checker as detailed in Section 9.5. The summarised DCM shown in Table 9.2 was crucial in order to refine the *clr* function. The RDQA was performed on a 32 bits 2.2Ghz CPU in about 20 minutes using 100 MB of RAM. Results are shortly commented in the following list:

Case 1: represents careers *already clean* that have been left *untouched*, which are about 45% of the total.

Case 2: refers to careers considered (by *ccheck*) valid before but not after cleansing, although they have not been touched by *clr*. As expected this subset is empty.

Case 3: describes valid careers that have been *improperly changed* by *clr*. Note that, despite such kind of careers remain clean after the intervention of *clr*, the behaviour of *clr* has been investigated to prevent that the changes introduced by *clr* could turn into errors in the future.

Case 4: represents careers originally valid that *clr* has made invalid. These careers have proven to be very useful to identify and correct bugs in the *clr* implementation.

Case 5: refers to careers considered (by *ccheck*) not valid before but valid after cleansing, although they have not been touched by *clr*. Though the number of careers is negligible, this result was useful to identify and repair a bug in *ccheck*.

Case 6: describes invalid careers, that *clr* was able neither to detect nor to correct, and consequently they were left untouched.

Case 7: describes the number of (originally) invalid careers which *ccheck* recognises as *properly cleansed* by *clr* at the end.

Case 8: represents careers originally invalid which have been *not properly cleansed* since, despite an intervention of *clr*, the function *ccheck* identifies them still as invalid.

The DCM shows that the original database had a very low quality of data (only 45% of the original careers were not affected by consistency issues), therefore justifying the need for data cleansing. Furthermore, considering the single DCM entries, cases 3, 4, 6, and 8 provided useful information for improving the *clr*, while case 5 provided information for improving the *ccheck*. In summary the *ccheck* was used to check the *clr* and the *clr* was used to check the *ccheck* (here comes the name *double check*). Since both *ccheck* and *clr* implementations cannot be guarantee as error free, there is no assurance that all the possible errors could be found through the DCM, nevertheless it has helped to improve the cleansing routines in the just introduced industrial example.

As a natural extension of this work, we are exploiting formal methods to carry out sensitivity analysis on dirty data, i.e. to identify to what extent the dirty data, as well as the cleansing routines, may affect the value of statistical indicators that are computed upon the cleansed data.

Table 9.2: The Double Check Matrix on an administrative database

	Conditions			*Result*
Case	$ccheck(S_i)$	$equals(S_i, C_i)$	$ccheck(C_i)$	Cardinality
1	0	0	0	$96,353$
2	0	0	1	0
3	0	1	0	$32,789$
4	0	1	1	$1,399$
5	1	0	0	3
6	1	0	1	40
7	1	1	0	$74,904$
8	1	1	1	$8,078$

```
type
 workerId: 0..maxPId;
 eventId: 0..maxEId;
 dataType: 0..maxDId;
 EventType: Enum{st,ex,cn,cs};
const
 max_worker: MAX;
 min_worker: 0;
var
 worker: workerId;
 no_errors: boolean; catchs SQL exceptions
 index: workerId;
 max_PT: boolean;  -- default = false
 max_FT: boolean; -- default = false
 double_FT_PT: boolean; -- default = false
 event_iterator: workerId;

-- retrieve data from database for each worker in window
externfun startstate_call(): boolean "external.h" ;
-- stores data from database for each worker in window
externfun set_path(i:workerId ; p:workerId): boolean;
-- returns ID of worker "p"
externfun get_worker(p:workerId): workerId ;
-- returns the pointer to the first data record of worker "p"
externfun get_first_index(p:workerId): workerId;
-- returns true if exists an event for worker "p" to analyse, false otherwise
externfun exists_an_event(p:workerId): boolean;
-- returns true if exists a pair<first,second> of events for worker "p"  which
 invalidates the career
externfun safe_transition(p:workerId ; type:EventType): boolean;

ruleset p:min_worker..max_worker do
  startstate "start"
  BEGIN
   no_errors := startstate_call();
   worker := get_worker(p);
   event_iterator := get_index(worker);
   no_errors := set_path(event_iterator,p);
  END;
END;

rule "next_event"
exists_an_event(worker)==>
BEGIN
 event_iterator := event_iterator + 1;
END;

invariant "Valid Start Event"
safe_transition(event_iterator,"st");
invariant "Valid Extension Event"
safe_transition(event_iterator,"ex");
invariant "Valid Conversion Event"
safe_transition(event_iterator,"cn");
invariant "Valid Cessation Event"
safe_transition(event_iterator,"cs");
invariant "Part-time"
max_PT = false;
invariant "Full-time"
max_FT = false;
invariant "Part-time and Full-time"
double_FT_PT = false;
```

Figure 9.4: The CMurphi model of *The Worker Career Administrative Archive* application.

```
--Invalid Career No 00005----------------
--Startstate (0000072)----------------

  Worker  EventIterator   Date          Type      ContractType     Company Code

       73      282       37061          1         F               23680
       73      283       38149          4         F               23680
       73      274       38471          1         P               9165
       73      275       38471          4         P               9165
       73      276       38474          1         P               9165
(-->) 73       277       38482          4         F               9165
******************** (Valid Cessation Event) failed
       73      278       38506          1         F               9165
       73      279       38509          4         F               9165
       73      280       38570          1         F               9165
       73      281       38572          4         F               9165
       73      284       38589          1         F               40617
       73      285       39591          4         F               40617
       73      286       39592          1         F               40617
---------------------------------------
```

Figure 9.5: Example of invalid career given by CMurphi.

Part III

Explicit Model Checking for the Analysis of Non-deterministic Systems

The last part of the Thesis is devoted to the analysis of systems having a non-deterministic behaviour. In Chapter 10 we discuss the problem of synthesising *strong* plans, and we provide a survey on the state of the art in this field. Then, in Chapter 11 we propose a novel algorithm to synthesise *optimal* strong plans for non-deterministic systems. Finally, in Chapter 12 some case studies on which we applied the algorithm are presented.

CHAPTER 10

STRONG PLANNING FOR NON-DETERMINISTIC DOMAINS

10.1 INTRODUCTION AND RELATED WORK

In recent years, a mutual interest between control theory and AI planning communities has emerged, showing that planning and control are closely related areas. The use of sophisticated controllers as well as intelligent planning strategies has become very common in robotics, manufacturing processes, critical systems and, in general, in hardware/software embedded systems (see, e.g., [33]).

In particular, efforts made to deal with planning in non-deterministic domains could be very helpful to solve control problems for real-world appliances. Indeed, many processes take place in an environment that may have variable and unpredictable influences on the action outcomes, which need to be taken into account to design a correct and efficient control system.

Informally speaking, non-deterministic domains model a particular form of uncertainty on system's action. More precisely, actions may have different outcome that is unpredictable at planning time. Indeed, given an action, it is impossible for the planner to know a priori which the outcome will be. In such context the concept of *plan*, which is well defined for deterministic planning, become ambiguous for non-deterministic system (e.g., the plan reaches the goal *always*? The plan reaches the goal *sometimes*?). In order to clarify this concept, the community distinguishes three possibilities, as introduced in [36]:

Weak Plan: It is a plan that may achieve the goal, but no guarantees are given about its success. More precisely, at least one of the many non-deterministic plan executions reaches the goal.

Strong Plan: It is a plan that is guaranteed to achieve the goal regardless of non-determinism. In other words, any plan execution always leads to a goal state.

Strong Cyclic Plan: It is a plan that achieves the goal under the *fairness* assumption. At execution time the plan applies an action infinitely often until the "lucky" outcome happens (i.e., the action's outcome that leads the system towards the goal). If it does not happen then the execution is called *unfair*.

Intuitively, weak solutions represent *optimistic* plans, i.e. we hope that the plan execution will reach the goal. On the contrary, strong solutions represent a *warranty* about the success of the plan, in any execution. Finally, strong cyclic solutions, which are based on a trial-and-error strategy, represent an intermediate alternative between strong and weak ones.

Moreover, another important aspect about planning, and then also for planning in non-deterministic domains, focuses on system's variables (e.g., can we observe all variables' values *before* the system execution? Some values are available only at run-time?). More precisely, we distinguish between two kinds of planning contexts:

Full-observability. All variables' values are available before the run-time. The generated plans are executed by a reactive controller that iteratively senses the world, determines the current state, and then it selects and executes an appropriate action (see, e.g., [109]).

In this setting, as typically in the case of dynamic systems, the size of the graph defining the dynamics of the system is exponential (*state explosion*) in the size of the input. As a result, classical algorithms for explicit graphs cannot be used, and instead suitable symbolic (e.g., [30]) or explicit (e.g., [54]) algorithms are used to counteract state explosion. This is also the typical situation for model checking problems.

Partial-observability. It works on a setting where only a subset of variables are observable and the remaining ones are available at run-time (see, e.g., [17, 18, 103]). A case limit of this context is the conformant planning [24, 4] were no observation is available, even at run-time.

In this section we focus on the synthesis of *strong solutions* in full-observability context.

In the Part II of the Thesis we discussed the problem of universal plans, whilst in this part we address the problem of strong planning. For the sake of clarity, we have to note that strong plan and universal plan, in a non-deterministic context are closely related, as both approaches aim to find a path to a goal from any state reachable from the initial ones. Finally, it is worth noting that systems may have or not probabilities associated to the actions outcomes. In our context, we focus on systems in which no probabilities are given.

Planning based on MDPs has been proved very effective (see, e.g., [27, 25, 170]), and, more recently, a variety of techniques have been proposed to solve continuous MDPs (see, e.g., [130, 127, 126]). However, MDP-based approaches deal with probabilistic distributions taking into account the stochastic outcomes of actions. Therefore, whether a solution provided by MDP planning algorithms is strong depends on probability and cost distribution. There is also a link between strong planning and CTL model checking. Indeed, it has been observed [107, 111] that the problem of strong and strong cyclic solution for a planning problem can be also addressed using the CTL model checking.

More precisely, applying the same idea of planning via model checking we described before, the search of a strong cyclic goal φ can be casted as the problem to satisfy the CTL formula $AGEF\neg\varphi$. Similarly, the synthesis of a strong solution can be solved satisfying the formula $AF\neg\varphi$ [45, 143] (see Section 4.1 for the semantics of CTL formulae).

In the last years, the (strong) planning in non-deterministic domains has had a growing interest in the planning community and then many techniques [115, 112, 111] and heuristics [78, 125] have been proposed.

A key contribution in this field comes from [36], where the authors present an algorithm to find strong plans implemented in MBP, a planner based on symbolic model checking. MBP produces a universal plan [150] which provides optimal solutions with respect to the plan length (i.e., the worst execution among the possible non-deterministic plan executions is of minimum length). Moreover, the use of Ordered Binary Decision Diagrams (OBDDs) together with symbolic model checking techniques allows a compact encoding of the state space and very efficient set theoretical operations.

More recently, in [112, 111] Kissmann and Edelkamp improved MBP by developing *Gamer*, a BDDs based planner able to synthesise strong and strong cyclic solutions for non-deterministic domains. In particular, Gamer works as follows: (1) it transforms the non-deterministic planning problem into a two-players turn-taking using a non-deterministic version of the PDDL language (i.e., NPDDL [19]). To give the main idea, the first player chooses an action (i.e., it sets the value of a variable) and the second player (i.e., the behaviour) chooses one of the possibile outcome for the action. Clearly, the translation process guarantees that the second player will choose all the possible non-deterministic behaviour for the selected action. Then, a minimal state encoding for the domain is computed resulting in a minimised state encoding [64] which provides a very compact BDD. After the translation it (2) applies a modified version of the algorithm by Cimatti et Al [36] which is able to deal with the two players behaviour. Despite a little overhead to enlarge the state definition in supporting of players' turn, the main benefit given by the translation process is to lead to a small BDD, and then to a better performance, with respect to MBP. This approach, which is very promising, needs to be further tested on many other non-deterministic domains.
However, both MBP and Gamer are backward-search planners based on a symbolic approach and then they requires to compute the inverse function of the system dynamics, which may be difficult to invert typically for nonlinear domains. Indeed, explicit algorithms (that explore forward the dynamics graph) allow to handle hybrid and/or nonlinear continuous domains, which are actually very common in the practice. On the other hand, explicit algorithms can generate a huge transition graph which may anticipate the state explosion.

In [115] the NDP algorithm exploits the use of some classical planners (e.g., FF [91] and SGPlan [101] are used in instance) to deal with non-deterministic domains, looking for strong cyclic solution in full observable domains. Given a planning problem P and a classical planner R, the algorithm generate a *deterministic* subproblem of P (e.g., if an action a has two distinct outcomes then it replaces action a with two new actions a_1, a_2

having the two outcomes respectively). Then, it call R on each sequence of classical subproblem. Finally, it collects and combines the $R's$ results for each sequence providing the final solution, if any. To compact encode domain literals, it uses the *conjunctive abstraction* instead of BDDs. The former has a compression efficiency less powerful than BDDs, nonetheless it can be used by any classical planner which deal with STRIPS domain, avoiding any modification to the planner.

Moreover, also heuristics have been exploited in combination with BDDs in order to improve the performance of such planners in real-world domains. To give an example, UMOP [107] uses both BDDs and heuristics to *guide* the search for strong and strong solutions. An improvement of UMOP has been presented recently in [106] where authors introduce NADL (a new description language for non-deterministic domains). Moreover, the new UMOP planner provides an algorithm (based on the previous one) which looks for *optimistic* plans by relaxing domain optimality constraints when no strong and strong cyclic solutions exist. In [125] a LAO* search (based on AND/OR graph) with heuristics based on pattern databases [67] have been used to synthesise strong and strong cyclic solutions.

In the following we provide a description MBP which, among the several approaches dealing with non-determinism, represents a milestone

10.2 THE MBP STRONG PLANNING PROCEDURE

Since our work about non-deterministic planning is inspired by the work of Cimatti et al, in this section we briefly describe the *strong planning procedure* as developed in [36]. As said previously, the algorithm of Cimatti et al. works on a *symbolic* representation of the state space, which we have shortly introduced in Chapter 4. To avoid the introduction of new formalisms, we adapt the STRONGPLAN procedure to works on the definitions of NDFSS as given in Sections 2.2 and 2.3.

Then, generally speaking, given a NDFSS according to Definition 6, and a set G of goals, a strong solution is a collection of policies (or a *state action* table) that maps a state $s \in Reach(S)$ to an action $a \in \mathcal{A}$ if s has a strong plan (i.e., if s always reaches a goal state in spite of the uncertainty of action a).

Procedure 12 shows the STRONGPLAN algorithm implemented in MBP planner. The procedure is a breadth first search which goes backward from goals states towards initial ones. At each step the solution of the previous step $OldSolution$ (i.e., the state action table) is updated and the algorithm stops when (1) either the solution can not be further updated or an initial state is reached (condition of line3).
During the iteration the algorithm needs to compute the inverse function of the dynamics graph due to the backward search. This work is accomplished by the function STRONG-PREIMAGE. More precisely, it is defined on the system state space as follows :

$$\text{STRONGPREIMAGE(S)} = \{(s,a) \text{ such that } \emptyset \neq F(s,a) \subseteq S\}$$

Intuitively, the STRONGPREIMAGE(S) computes the set of state-action (s,a) pairs which guarantees that all the reachable states from s via action a belong to S, in spite of the non-determinism (i.e., $F(s,a)$ is a set of feasible states for the system). However, during the backward visit it is possible to consider a pair (s,a) for with a solution is already known for s. It is not a negligible aspect since it may affect the minimality of the final solution, that is, the strong plan may contain more different actions to apply in the same state. To this aim, the function PRUNEDSTATE scans the preimage table by removing all the pairs (s,a) which start from the same state s.

Finally, if a final solution exists then the algorithm returns it, otherwise the algorithm returns $fail$.

Procedure 12 STRONGPLAN

```
1:  $OldSolution \leftarrow \emptyset$ //  Solution of the previous step
2:  $Solution \leftarrow \emptyset$ // Solution of the actual step
3:  while ($OldSolution \neq Solution \wedge I \not\subseteq (G \cup Solution)$) do
4:    $PreImage \leftarrow$STRONGPREIMAGE($G \cup Solution$) //  Compute the inverse function of the dynamics
5:    $NewSolution \leftarrow$PRUNEDSTATES($PreImage, G \cup Solution$)
6:    $OldSolution \leftarrow OldSolution \cup NewSolution$
7:  end while
8:  if $I \subseteq (G \cup Solution)$ then
9:    return $Solution$;
10: else
11:   return $fail$;
12: end if
```

10.3 WORKING EXAMPLE

The following example (as presented in [37]) should help to clarify the concept of strong plan. In their article, Cimatti et al. use the $\mathcal{AR}$ language to describe the domain, however, to avoid the introduction of new formalisms, we use a graphical representation of the domain as in Figure 10.1.

A pack can be moved from a *London Heatrow* city airport to one of *Gatwick* or *Luton* airports via railway, truck or airplane. The state is composed by 4 different variables: *pos* ranges in {*train-station, truck-station, air-station, Victoria-station, city-center, Gatwick, Luton* }. Variables $fuel$ and fog are boolean whilst the variable $light$ ranges in $\{green, red\}$. Moreover, when a state variable value is omitted then it could have any value in its range. The possible actions are *drive-train*, *wait-at-light*, *drive-truck*, *make-fuel*, *fly* and *air-truck-transit*. Note that, as depicted in Figure 10.1, actions *drive-truck* and *drive-train* may be non-deterministic (e.g., action *drive-train* executed in position *train-station* may lead to two different outcomes: at Victoria's station with *red* or *green* traffic light). The STRONGPLAN procedure performed on this domain produces a solution as shown in Table 10.1.

Roughly speaking, the solution is a *state-action* table that summarises all the states able to reach the goal, in spite of non-determinism. For each of such states it suggests the *strong* action to apply to reach the goal. Note that, since no weights on actions are given (i.e., all transitions impliedly have a weight equals to 1) a backward BF visit, which is performed by the STRONGPLAN, always returns an optimal solution with respect to the distance to the goal.

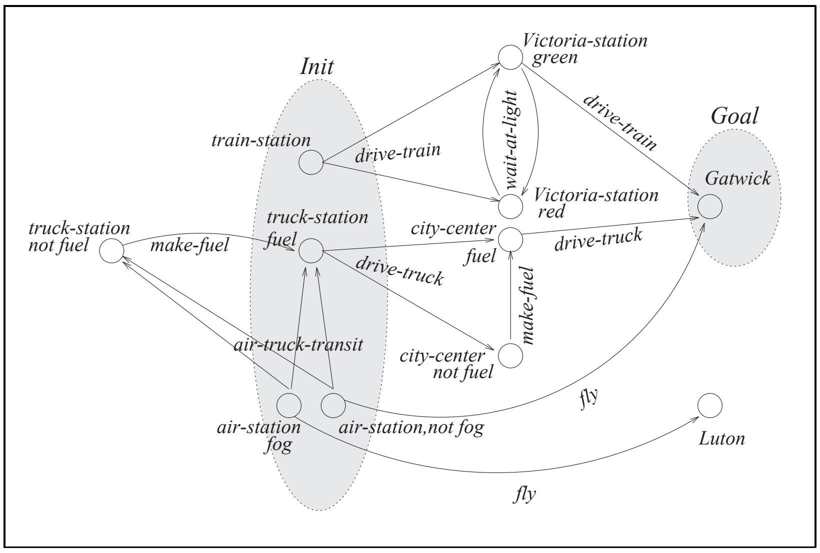


Figure 10.1: Informal description of the working example domain.

Table 10.1: A Strong Plan for the domain of Figure 10.1.

State	*Action*	*Distance to Goal*
pos =*Victoria-Station* ∧ *ligth* =*green*	*drive-train*	1
pos =*City-Center* ∧ *fuel* =*true*	*drive-truck*	1
pos =*Air-Station* ∧ *fog* =*false*	*fly*	1
pos =*Victoria-Station* ∧ *ligth* =*red*	*wait-at-light*	2
pos =*City-Center* ∧ *fuel* =*false*	*make-fuel*	2
pos =*Train-Station*	*drive-train*	3
pos =*Truck-Station* ∧ *fuel* =*true*	*drive-truck*	3
pos =*Truck-Station* ∧ *fuel* =*false*	*make-fuel*	4
pos =*Air-Station* ∧ *fog* =*false*	*air-truck-transit*	4

CHAPTER 11

STRONG PLANNING THROUGH EXPLICIT MODEL CHECKING

11.1 INTRODUCTION AND CONTRIBUTION

In Chapter 10 we introduced the Strong Planning problem in non-deterministic domains, focusing on the state of the art in this field. Moreover, we briefly described the approach of MBP and Gamer planners, which represent a milestone in the context of strong planning.

In this chapter we propose an algorithm (whose a preliminary version has been published in [53]) to solve the *cost-optimal* strong planning problem in non-deterministic FSSs (according to definitions given in Section 2.3). Roughly speaking, we are interested in finding strong solutions having a *minimum* cost with respect to a given cost function. Our algorithm is strictly based on [37] (which we described in Section 10.2), however, there are the following main contributions:

1. we consider a *cost function* and present a novel technique to look for *cost-optimal* strong plans while preserving a good complexity bound.
2. we use an *explicit approach* rather than a symbolic one, so extending the class of problems on which strong planning can be applied to hybrid and/or nonlinear continuous domains, which are actually very common in the practice. Indeed, since representing addition and comparison with OBDDs requires opposite variable orderings and since this kind of problems involve both such operations, OBDDs in our context tend to have size exponential in the input size. On the other hand, using an explicit approach allows us to expand "on demand" the transition relation, generating and representing only the reachable states.

Finally, in Sections 11.6 and 11.7 we formally prove the correctness, the completeness and the complexity of the proposed algorithm. Furthermore, in the next Chapter we present some experimental results showing the effectiveness of the proposed approach on two meaningful case studies. In order to do this, we define the concept of cost-optimal strong plan and the corresponding planning problem.

For the sake of clarity, it is worth noting that the algorithm in [37] could be adapted to support costs and devise cost-optimal solutions only by ”unary encoding” the weights, i.e., by replacing a transition of weight k with k contiguous deterministic transitions. In this case, however, its complexity, in the worst case, would be exponentially higher than the one of the algorithm presented in the following.

11.2 RELATED WORK

The cost-optimal strong planning problem could also be cast as a strategy synthesis problem for a multistage game with two players moving simultaneously (see, e.g., [79]), where the first player is the controller, the second is the disturbance (causing the non-deterministic behaviours), and the game rules are given by the plant dynamics. In this setting, our control strategy could be seen as a *minmax* strategy for the controller player. That is, in each game state, the controller chooses the action that minimises the maximum cost (to reach the goal) that the disturbance, with its (simultaneous) choice, may inflict to it. Such a game theoretic casting, however, would be of little help from a computational point of view, since in our setting the normal form of the game would be intractable even for small systems. Indeed, if the game has $|S|$ states and, in each state, the controller and the disturbance have at most $|\mathcal{A}|$ and $|D|$ actions available, respectively, then the game would be represented by a graph with $|S|$ nodes, each having $|\mathcal{A}||D|$ outgoing edges. Thus, even considering simple plants, we would have very large graphs (see the Chapter 12 on case studies).

The situation is exactly analogous to that for model checking based analysis of Markov chains (e.g, see [116], [55]). Of course, in principle, stationary distributions for Markov chains can be computed using classical numerical techniques (e.g., see [14]) for Markov chains analysis. However, for dynamic systems, our setting here, the number of states (easily beyond 10^{10}) of the Markov chains to be analysed rules out matrix based methods.

Finally, casting our problem as a Mixed Integer Linear Programming (MILP) problem would be possible but, again, it would generate a MILP of size exponential in the input. Thus, to the best of our knowledge, this is the first approach to cost-optimal strong planning and no better solutions for this problem have been devised so far, even in other computer science fields.

11.3 COST-OPTIMAL STRONG PLAN

In order to discuss the problem to find cost-optimal trajectories, we extend our setting with a cost function.

Definition 17 (Cost Function)**.** *Let* $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$ *be an* NDFSS. *A* cost function *(also called* weight function*) for* $\mathcal{S}$ *is a function* $\mathcal{W} : \mathcal{S}_\tau \to \mathbb{R}_+$ *that assigns a cost to* *each transition in $\mathcal{S}$. Using the cost function for transitions, we define the cost of the non-deterministic transition $(s,a,F(s,a))$, denoted by $\mathcal{W}(s,a)$, as follows: $\mathcal{W}(s,a) = \max_{s' \in F(s,a)} \mathcal{W}((s,a,s'))$.*

It is worth noting that, for the sake of generality, the definition of the cost function above allows the transition cost to depend on both the corresponding action and the source state. However, usually the transition costs are bound to the corresponding action only.

Now let $\mathcal{S}$ be a given *NDFSS* according to Definition 6. In order to define the cost-optimal strong planning problem for such a kind of system, we assume that a non-empty set of *goal states* $G \subset S$ has been specified. Then, a Cost-Optimal Strong Planning Problem (COSPP) can be defined as follows.

Definition 18. *(Cost-Optimal Strong Planning Problem) Let $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$ be a* NDFSS. *Then a* Cost-Optimal Strong Planning Problem (COSPP) *is a triple $\mathcal{P} = (\mathcal{S}, \mathcal{W}, G)$ where G is the set of the goal states and $\mathcal{W} : \mathcal{S}_\tau \to \mathbb{R}_+$ is the cost function associated to $\mathcal{S}$.*

In this setting, we aim to find a *strong plan* from the initial state s_0 to G, that is a sequence of actions that, starting from s_0, leads the system to the goal states, regardless of the non-deterministic outcome of each action. Before formally describe such a solution, we need to define the structure of a deterministic plan.

Definition 19. *(Deterministic Plan) Let $\mathcal{P} = \{\{S, s_0, \mathcal{A}, F\}, \mathcal{W}, G\}$ be a* COSPP *and $s \in S$. A* deterministic plan *p from s to a goal $g \in G$ is a trajectory π such that:*

- *either $s \in G$ and $|\pi| = 0$;*
- *or $\pi = \tau_0, \tau_1, \ldots, \tau_n$, with $\tau_0 = (s,a,s_1)$ and $\pi' = \tau_1, \ldots, \tau_n$ is a* deterministic plan *from s_1 to g.*

In other word, we define a deterministic plan as a sequence of extracted transitions, according to Definition 9. Intuitively, a cost-optimal strong solutions is a set of non-deterministic transitions as stated in Definition 7. More precisely, we are interested in finding a cost-optimal solutions defined as follows.

Definition 20. *(Strong Plan) Let $\mathcal{P} = \{\{S, s_0, \mathcal{A}, F\}, \mathcal{W}, G\}$ be a* COSPP. *Let s be a state in S. A* strong plan *from s to G is a set P of non-deterministic transitions such that either $s \in G$ and $P = \emptyset$ or $s \notin G$ and P satisfies the following conditions:*

1. *there exists a natural number n_0 such that every trajectory π that can be extracted from P has length $|\pi| \leq n_0$;*
2. *every trajectory π starting from s which can be extracted from P, can be extended to a deterministic plan π' from s to a goal $s_g \in G$ such that π' is extracted from P;*

3. *for every state s' such that $s' \notin G$ and s' is in P there exists a trajectory π, extracted from P, starting from s and ending in s';*

4. *for every state s' such that $s' \notin G$ and s' is in P, there exists exactly one non-deterministic transition in P of the form $(s', a, F(s', a))$, for some $a \in \mathcal{A}$. We denote with $P(s')$ such non-deterministic transition.*

We have the following characterisation of plans.

Proposition 3. *Let $\mathcal{P} = \{\mathcal{S}, \mathcal{W}, G\}$ be a* COSPP. *P is a strong plan from s to G iff P is a set of non-deterministic transitions such that either $s \in G$ and $P = \emptyset$ or $s \notin G$ and there exists a unique non-deterministic transition in P of the form $\tau = (s, a, F(s, a))$, for some $a \in \mathcal{A}$, such that:*

- *either $F(s, a) \subseteq G$;*
- *or $P \setminus \{(s, a, F(s, a))\}$ is the union of strong plans P_i from every state s_i in $F(s, a)$ to G.*

Proof. Assume first that P is a strong plan from s to G. If P is not empty, then there exists a unique non-deterministic transition $\tau = (s, a, F(s, a)) \in P$, for some a. Let s_i be an element of $F(s, a)$. We define P_i as the set of non-deterministic transitions in P such that P_i contains some transitions of a deterministic plan from s_i.

Now observe that any sequence starting from s_i can be completed in P to a deterministic plan without using the node s. Indeed, no deterministic plan extracted from P can return to the node s, since otherwise there would be a cycle, contradicting the requirement that every deterministic sequence in P is bounded. It follows that P_i is a subset of $P \setminus \{\tau\}$ and is a strong plan from s_i.

Moreover, let s' be any node in $P \setminus \{\tau\}$. Then there exists a trajectory π from s to s'. By the uniqueness of τ, the first transition of π is in τ and therefore has the form (s, a, s_i) for some s_i, it follows that s' is in P_i, and that $P \setminus \{\tau\} = \bigcup_{s_i \in F(s,a)} P_i$.

The other direction is easy and left to the reader. □

By Proposition 3 we can define the cost of a plan as follows:

Definition 21. *(Strong Plan Cost) The cost of a strong plan P from s to G, denoted by $\mathcal{W}(P)$, is defined by recursion as follows:*

- *if P is empty then $\mathcal{W}(P) = 0$;*
- *if P is composed only of the non-deterministic transition $(s, a, F(s, a))$, for some a, then $\mathcal{W}(P) = \mathcal{W}(s, a)$;*

- *if P is composed of the non-deterministic transition $(s, a, F(s, a))$, for some a, and of plans P_i from every node s_i in $F(s, a)$ then $\mathcal{W}(P) = \max_{s_i \in F(s,a)} (\mathcal{W}((s, a, s_i)) + \mathcal{W}(P_i))$.*

It is easy to see that the cost of a plan P is the maximum cost of a deterministic plan extracted from P.

Definition 22. *(Minimum Strong Plan Cost) Let $\mathcal{P} = \{\mathcal{S}, \mathcal{W}, G\}$ be a* COSPP, *with $\mathcal{S} = \{S, s_0, \mathcal{A}, F\}$. Then a* cost-optimal strong solution *of the* COSPP *$\mathcal{P} = \{\mathcal{S}, \mathcal{W}, G\}$ is a strong plan P from s_0 to G such that the cost of P is minimal among the strong plans from s_0 to G.*

11.4 An Example of Cost-Optimal Strong Planning Problem

As an example of COSPP, let us consider the *hurried passenger* problem. A passenger wants to arrive to San Francisco airport (SFO) departing from one of the Rome airports (CIA or FCO) and according to the flight scheduling shown in Table 11.1. Moreover, there is a bus on every hour that allows the passenger to go from home to one of the Rome airports above in one hour.

Table 11.1: Flight scheduling.

From	To	Flight #	Depart	Arrive
Rome-FCO	Paris-CDG	A	08.00	09.00
Rome-FCO	Berlin-BER	E	08.00	10.00
Rome-CIA	Amsterdam-AMS	D	05.00	08.00
Paris-CDG	San Francisco-SFO	B	10.00	12.00 (GMT-7)
Paris-CDG	San Francisco-SFO	C	19.00	21.00 (GMT-7)
Berlin-BER	San Francisco-SFO	F	11.00	14.00 (GMT-7)
Berlin-BER	Amsterdam-AMS	I	12.00	13.00
Berlin-BER	San Francisco-SFO	G	12.00	15.00 (GMT-7)
Amsterdam-AMS	San Francisco-SFO	H	15.00	20.00 (GMT-7)

The goal is to arrive to San Francisco as soon as possible, and, however, no later than 21.00 local time. We require the passenger to arrive at the airport at least one hour before a flight departure. Moreover, we assume that each flight may arrive at destination later than the expected arrival time. The objective is to generate a strong plan (if any) that guarantees

Table 11.2: COSPP for the *hurried passenger* problem.

S	home = s_0, AMS = s_1, AMS$_d$ = s_2, CDG = s_3, CDG$_d$ = s_4, CIA = s_5, FCO = s_6, BER = s_7, BER$_d$ = s_8. SFO$_a$ = s_9, SFO$_m$ = s_{10}, SFO$_n$ = s_{11}.
$\mathcal{A}$	A, B, C, D, E, F, G, H, I, P, Q
F	$F(s_0,Q) = \{s_6\}$, $F(s_0,P) = \{s_5\}$ $F(s_1,H) = \{s_9\}$, $F(s_2,H) = \{s_9\}$ $F(s_3,B) = \{s_9,s_{10}\}$, $F(s_4,C) = \{s_9,s_{11}\}$ $F(s_5,D) = \{s_1,s_2\}$ $F(s_6,A) = \{s_3,s_4\}$, $F(s_6,E) = \{s_7,s_8\}$ $F(s_7,F) = \{s_9\}$, $F(s_8,G) = \{s_9\}$, $F(s_8,I) = \{s_1,s_2\}$
$\mathcal{W}$	$\mathcal{W}(s_0,Q,s_6) = 1$, $\mathcal{W}(s_0,P,s_5) = 1$; $\mathcal{W}(s_1,H,s_9) = 12$, $\mathcal{W}(s_1,H,s_9) = 13$; $\mathcal{W}(s_2,H,s_9) = 11$, $\mathcal{W}(s_2,H,s_9) = 12$; $\mathcal{W}(s_3,B,s_{10}) = 10$, $\mathcal{W}(s_3,B,s_9) = 11$; $\mathcal{W}(s_4,C,s_9) = 18$, $\mathcal{W}(s_4,C,s_{11}) = 19$; $\mathcal{W}(s_5,D,s_1) = 9$, $\mathcal{W}(s_5,D,s_2) = 10$; $\mathcal{W}(s_6,A,s_3) = 2$, $\mathcal{W}(s_6,A,s_4) = 3$, $\mathcal{W}(s_6,E,s_7) = 3$, $\mathcal{W}(s_6,E,s_8) = 4$; $\mathcal{W}(s_7,F,s_9) = 11$, $\mathcal{W}(s_7,F,s_9) = 12$; $\mathcal{W}(s_8,G,s_9) = 11$, $\mathcal{W}(s_8,G,s_9) = 12$, $\mathcal{W}(s_8,I,s_2) = 3$, $\mathcal{W}(s_8,I,s_1) = 2$;
G	s_9, s_{10}
P	$P(s_0) = Q(17); P(s_5) = D(22); P(s_6) = E(16); P(s_7) = F(12)$; $P(s_8) = G(12); P(s_2) = H(12); P(s_1) = H(13)$, $P(s_3) = B(11)$

the passenger to reach the San Francisco airport before 21.00 local time regardless of possible flight delays.

The corresponding COSPP (according to Definitions 6,17 and 18) is reported in Table 11.2. Here the actions correspond to the flights and the non-determinism is given by the possible delay which, for the sake of simplicity, we assume to be limited to one hour for each flight. The cost of each transition (d, f, a) is $\mathcal{W}(d, f, a) = (t(d) + t(f) + t(a))$ where $t(d)$ is the time spent at airport d waiting for the flight departure, $t(f)$ is the duration of the flight and $t(a)$ is the time spent at airport a waiting for the next flight (which could be zero). Moreover, the special actions P and Q represent the bus journey from home (state s_0) to Rome-CIA and Rome-FCO, respectively: for the sake of simplicity, we do not consider delays on these actions, so the corresponding transitions are deterministic and have cost 1 (i.e., the bus journey takes a hour).

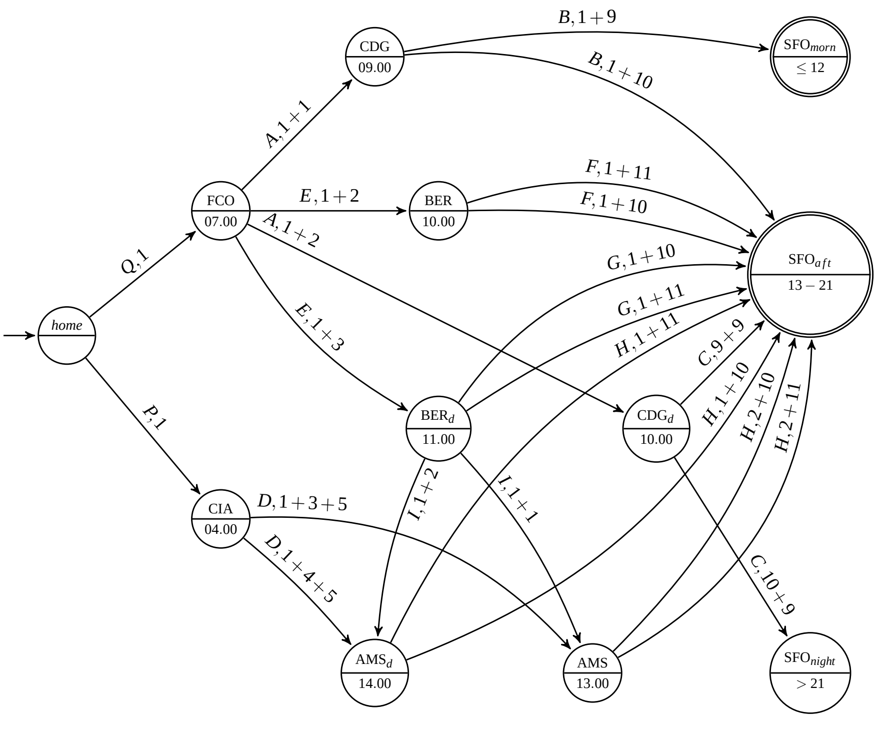


Figure 11.1: Graphical description of the COSPP for the *hurried passenger* problem.

A graphical description of the problem is given in Figure 11.1 where tagged nodes represent the arrival time at the corresponding airport, while the edges are labelled with the flight code.

The cost-optimal solution consists in flying from Rome-FCO to Berlin-BER and then to San Francisco-SFO. The total cost of the solution (considering all the possible delays) is 17. Note that another strong solution would be flying from Rome-CIA to Amsterdam-AMS and then to San Francisco-SFO, but its cost is 23. Finally, flying from Rome-FCO to Paris-CDG is not a strong solution since in case of delay of flight A it would be impossible to reach San Francisco on time.

11.5 The Cost-Optimal Strong Plan Algorithm

In this section we describe a procedure that looks for a cost-optimal strong solution to a given COSPP. The main algorithm (Procedure 16) consists of two subroutines described in the following. All the procedures make use of some auxiliary functions and sets : *Cost*, *Cand*, *ExtGoals*, *OldExtGoals* and Δ .

The *cost vector* $(Cost)$ is used to maintain the set of states for which we have synthesised a strong plan, sorted by their cost. By abuse of notation, in the following we refer to $cost(s)$ as the *cost function* which returns the minimum cost of a strong plan from s to the goals calculated so far. The algorithm updates this function every time a better strong plan is found for s. Initially all the goal states have a cost equal to zero, while the cost of the other states is set to ∞.

The set of *candidates* (*Cand*) contains the pairs (s,a) corresponding to all the states s which, at any step, are recognised to have a plan starting with action a, possibly of *non* minimum cost. The elements in the set *Cand* can be partially ordered with respect to the cost function *Cost*. Initially the set *Cand* is empty.

The set of *extended goals* (*ExtGoals*) contains all the states s which, at any step, are recognised to have a plan P of *minimum cost*. Initially the set *ExtGoals* contains all the goal states in G. On the other hand, the set of *old extended goals* (*OldExtGoals*) contains, at any step, the extended goals collected up to the previous step: that is, the expression $ExtGoals \setminus OldExtGoals$ represents the states that have been just added to the extended goals.

The set $\Delta(s,a)$ is initialised, for each state-action pair, with the states reachable from s via action a, i.e., $F(s,a)$, which are consumed during the algorithm iterations.

In the following, we assume that all procedures take as input the COSPP $\mathcal{P} = ((S,s_0,\mathcal{A},F),\mathcal{W},G)$ as well as the auxiliary sets and functions. The output is a strong plan SP.

Note that, in the algorithms, some arithmetic operations (i.e., min, max and sum) may involve infinity. In this case, we assume the usual semantics, e.g., $\max(x,\infty) = \infty$ or $x+\infty = \infty$.

11.5.1 The CandidateExtension routine

The CandidateExtension routine (Procedure 13) extends the set *Cand* of candidates. The function $Pre(s)$ returns all the transitions leading to s and is applied to the extended goals found in the previous iteration of the main algorithm. At any step, the set $\Delta(s,a)$ contains only the states reachable from s via action a which have not been moved to the extended goals yet. Thus, once $\Delta(s,a)$ is empty, s is guaranteed to have a strong plan through action a, since all the transitions in $(s,a,F(s,a))$ lead to an extended goal. The pair (s,a) is then added to the set of candidates if it improves the cost currently associated to s.

Procedure 13 CANDIDATEEXTENSION

```
for all $s' \in (ExtGoals \setminus OldExtGoals)$ do
  $Pre(s') \leftarrow \{(s,a) \in S \times \mathcal{A} | s' \in F(s,a)\}$;
  for all $(s,a) \in Pre(s')$ do
    $\Delta(s,a) \leftarrow \Delta(s,a) \setminus \{s'\}$;
    if $\Delta(s,a) = \emptyset$ then
      $c' = \max_{\bar{s} \in F(s,a)}(\mathcal{W}(s,a,\bar{s}) + Cost(\bar{s}))$;
      if $c' < Cost(s)$ then
        $Cand \leftarrow Cand \cup (s,a)$;
        $Cost(s) = c'$;
      end if
    end if
  end for
end for
```

11.5.2 THE PLANEXTENSION ROUTINE

The effect of the PLANEXTENSION routine (Procedure 14) is twofold. First, it selects the states in the candidates set of minimum cost and moves them to the set of extended goals. Indeed, the current solution for such states cannot be improved, since there are no actions which provide a strong solution with a lower cost (see Proposition 5). Second, it inserts the new extended goals together with the associated action (i.e., the corresponding non-deterministic transition) in the strong plan *SP*.

Procedure 14 PLANEXTENSION

```
$\alpha \leftarrow \min_{(s,a) \in Cand} Cost(s)$; // It uses the MINCOSTCAND routine
for all $(s,a) \in Cand | Cost(s) = \alpha$ do
  $ExtGoals \leftarrow ExtGoals \cup \{s\}$;
  $Cand \leftarrow Cand \setminus \{(s,a)\}$;
  $SP \leftarrow SP \cup (s,a)$;
end for
```

Note that the extraction of the candidates with the lowest cost (first two lines of Procedure 14) can be accomplished with a small complexity if we suppose to have a structure $costvector$ where each element $costvector[c]$ holds a list of references to the states with cost c. Insertion in this structure is constant time, whereas updates can be also accomplished in constant time by re-inserting the state with updated cost without removing the previous instance (i.e., creating a duplicate with different cost). Indeed, the states with minimum cost can be extracted from this structure as shown by the MINCOSTCAND routine (Procedure 15).

Procedure 15 MINCOSTCAND

Input: *lastc*, the cost of the last states returned

```
1:  $c \leftarrow lastc$
2:  loop
3:    $c \leftarrow c+1$
4:    $AllCand_c \leftarrow costvector[c]$
5:    if $AllCand_c \neq \emptyset$ then
6:      $Cand_c \leftarrow \emptyset$
7:      for all $s \in AllCand_c$ do
8:        if $s \notin ExtGoals$ then
9:          $Cand_c \leftarrow Cand_c \cup \{s\}$
10:       end if
11:     end for
12:     if $Cand_c \neq \emptyset$ then
13:       $lastc \leftarrow c$
14:       return $Cand_c$
15:     end if
16:   end if
17: end loop
```

The procedure takes as input the cost of the last states returned, and scans the $costvector$ starting from the element corresponding to the next (higher) cost c (for the sake of simplicity, in the pseudocode we suppose it to be $lastc+1$, but in general it depends on the approximation of the cost function). If $costvector[c]$ contains some states that are not yet in the extended goals, the procedure returns them, otherwise it increases c and loops. Thus, even if updates may create duplicates of the same state in different elements of $costvector$, since the algorithm always extracts *first* the minimum cost instance of a state, and inserts it in $ExtGoals$, all its further instances (with higher cost) in $costvector$ will be simply ignored. However, from here on, we assume to use a Fibonacci's heap to maintain the $costvector[c]$.

11.5.3 THE COSTOPTIMALSTRONGPLAN ROUTINE

Finally, the COSTOPTIMALSTRONGPLAN routine (Procedure 16) initialises the cost value of each state and the sets Δ, $Cand$, $ExtGoals$ and $OldExtGoals$, then iterates applying the subroutines described above. In particular, the procedure loops until either the initial state s_0 is included in the extended goals (that is a strong solution has been found) or a fix point is reached, since there are no new extended goals (in this case there is no strong solution for s_0). Note that, as a collateral effect, the algorithm also finds all the strong plans for the states in S having minimal cost less or equal to the cost of s_0. Thus, if s_0 does not reach the goal (i.e., its cost is ∞), or if we explicitly remove the guard that stops the algorithm in this case, the COSTOPTIMALSTRONGPLAN would actually calculate a cost-optimal strong *universal* plan.

Procedure 16 COSTOPTIMALSTRONGPLAN

Input: a COSPP $\mathcal{P} = ((S, s_0, \mathcal{A}, F), \mathcal{W}, G)$

Output: a cost-optimal strong plan *SP*

```
for all $(s,a,s') \in \mathcal{S}_\tau$ do
  if $s \in G$ then
    $Cost(s) = 0$;
  else
    $Cost(s) = \infty$;
    $\Delta(s,a) = F(s,a)$;
  end if
end for
$Cand \leftarrow \emptyset$;
$SP \leftarrow \emptyset$;
$OldExtGoals \leftarrow \emptyset$;
$ExtGoals \leftarrow G$;
while ($ExtGoals \neq OldExtGoals$) do
  if $s_0 \in ExtGoals$ then
    return $SP$;
  end if
  CANDIDATEEXTENSION();
  $OldExtGoals \leftarrow ExtGoals$;
  PLANEXTENSION();
end while
return Fail;
```

11.6 Time Complexity of the Algorithm

In this section we first define a COSPP worst case instance an then we describe the algorithm behaviour on it.

Definition 23 (COSPP Worst Case Instance)**.** *Let $\mathcal{P} = \{\{S, s_0, \mathcal{A}, F\}, \mathcal{W}, G\}$ be a* COSPP *and let $h : F \rightarrow \mathbb{N}^+$ be a bijective function that associates to each transition $\tau \in F$ a unique natural number.*

1. *$\forall s_i, s_j \in S$ with $i \neq j, \forall a_l \in \mathcal{A}$ exists a transition of the form $\tau_{i,j,l} = (s_i, a_l, s_j)$. Therefore, $\forall s \in S, a \in \mathcal{A}$ we have $|F(s,a)| = (|S| - 1) \times |\mathcal{A}|$. Hence, the total number of transitions $|F|$ in the system is $\theta(|S|^2 \cdot |\mathcal{A}|)$.*
2. *$\forall \tau \in F, \mathcal{W}(\tau) = 10^{-h(\tau)}$.*
3. *G is the* minimal *goal set, i.e. $G = \{s_n\}$.*

Note that the instance in Definition 23 is a worst case instance for our algorithm since:

- constraint (1) guarantees that each action (regardless of its non-determinism) reaches *directly* any state (i.e., the graph is *complete* with respect to the non-determinism of each action). As a consequence each single state will have a strong plan;
- the usefulness of constraint (2) is twofold: it ensures that if the system has $|\mathcal{A}| = r$ distinct possible actions to reach a state s_j from s_i, then exists an ordered sequence of transitions $(s_i, a_1, s_j), \ldots, (s_i, a_r, s_j)$ where $\mathcal{W}(s_i, a_1, s_j) > \ldots > \mathcal{W}(s_i, a_r, s_j)$. The aim is to maximises the number of cost update that the algorithm performs during its execution. Furthermore, it guarantees that the cost of each trajectory from a state to a goal is unique in the whole system, and then the cost of each strong plan will be unique.

Proposition 4 (Algorithm Complexity)**.** *Let $|reach(S)| = n$ be the number of states in the system, let $|\mathcal{A}| = r$ be the number of action in the system and let $m = (n^2 \cdot r)$ be an upper bound to the maximum number of transitions in the system (i.e., one transition to reach each node for each action) the time complexity of the Algorithm is $O(m + n(r + \log n))$*

Proof. To prove the proposition we refer to the COSPP worst case instance according to Definition 23. During the proof, we assume to implement the *costvector* structure using the Fibonacci's heap.

At the first step the set *ExtGoals* is initialised with G states (i.e,$|ExtGoals|$ = 1). During the algorithm execution the sets *ExtGoals* and *OldExtGoals* will be always different between them. Indeed, at each step *exactly one* single state will be inserted into *ExtGoals*.

On the contrary the algorithm has reached its fixed point and then it stops performing less than n steps. Hence, in the worst case the algorithm requires at most n steps to find a strong solution.

It it clear that the algorithm complexity strongly depends on the complexities of CANDIDATEEXTENSION and PLANEXTENSION routines.

Looking at CANDIDATEEXTENSION routine of Procedure 13, it analyses one single state belonging to *ExtGoals* (as said previously) at each step (line 1). Since we assume to have in memory the expanded dynamics of the graph, it is not required to compute the *Pre* function of line 2, so this step is performed in constant time. Otherwise, we may apply an explicit state space exploration algorithm to build it in $O(|\mathcal{S}_\tau|)$. Looking at line 3, a single state $s \in Pre(s')$ can have at most $n \cdot r$ outgoing transitions. In other words, $|Pre(s')| = O(n \cdot r)$. Note that if it has fewer transitions then it must exist at least one state s unable to reach s', and this violates the hypothesis of Definition 23. However, the conditional statement starting at line 5 is performed only once for each call of the procedure. Roughly speaking, The CANDIDATEEXTENSION procedure removes one state from a set $\Delta(s,a)$ in each iteration. Indeed, the set $\Delta(s,a)$ requires at most n iterations of line 3 to become empty and then the $Cost(s)$ of line 6 is computed at most one for each action (i.e, at most r times for each call of the CANDIDATEEXTENSION). Moreover, the time needed to update the $Cost(s)$ of line 9 (which requires *insert* and/or *decrease key* operations) can be accomplished in constant time through Fibonacci's heap. Note that the operation *delete key* (which has a logarithmic complexity) is unneeded since the cost of a state (i.e. its key) is always nonincreasing.

Then, the cost of the CANDIDATEEXTENSION in the worst case is $O(n \cdot (n \cdot r + r)) = O(n^2 \cdot r + n \cdot r)$. Since the maximum number of transitions F of the system in the worst case is $m = O(n^2 \cdot r)$ the complexity of the routine is $O(m + n \cdot r)$. In the other words, the routine scans linearly the system graph and the complexity depends on how many time it needs to update the *costvector* of a single state.

Looking at Procedure 14, the PLANEXTENSION computes the α through a *Delete Min* operation in the cost vector, which requires $O(\log n)$ time using a Fibonacci's heap. Clearly, this operation requires to be performed at each iteration, i.e. at most n times in the worst case since all states will have a strong plan. Hence, the complexity of this procedure is $O(n \cdot \log n)$.

The overall complexity of COSTOPTIMALSTRONGPLAN is therefore $O(m + n(r + \log n))$. $\square$

Is worth to note that, in many real instances, the number of transitions for each state is enormously less than $n \cdot r$ which represents the maximum non-determinism degree of the system.

11.7 Correctness and Completeness of the Algorithm

The algorithm given in Procedure 16 essentially iterates the two procedures CANDIDATEEXTENSION and PLANEXTENSION until the desired state has a plan or the fix point is reached. Let us indicate with $ExtGoals_k$ and $Cand_k$ the contents of the $ExtGoals$ and $Cand$ sets, respectively, at the k-th step of the algorithm. Moreover, let us call $Goals_k$ the union $G \cup ExtGoals_k$.

Proposition 5 (Correctness)**.** *Let u_k be the maximum cost of a state in $Goals_k$, that is $u_k = \max_{s \in Goals_k} Cost(s)$. Then, in any step $k \geq 1$ of the* COSTOPTIMALSTRONGPLAN *algorithm, all the states with a plan of minimum cost no greater than u_k are in $Goals_k$. That is $\forall s \in S, Cost(s) \leq u_k \Rightarrow s \in Goals_k$*

Proof. At the first iteration of the algorithm ($k = 1$), $Goals_k = ExtGoals_k = G$ contains, by definition, all the states with a plan having cost zero (i.e., the goals).

Now, let us assume by induction that the property holds at step k. We shall prove that it still holds at step $k+1$, i.e., the new elements inserted in $Goals_{k+1}$ do not falsify it.

To this aim, let α_{k+1} be the minimum cost of a candidate in $Cand_{k+1}$, that is $\alpha_{k+1} = \min_{s \in Cand_{k+1}} Cost(s)$. We can simply prove that $u_k < \alpha_{k+1}$. Indeed, assume that $u_k \geq \alpha_{k+1}$: then there exists a state $s \in Cand_{k+1}$ s.t. $Cost(s) \leq u_k$. However, a state in $Cand_{k+1}$ cannot be in $Goals_k$ (since the algorithm moves to the $ExtGoals$ only states that are already in $Cand$), and this contradicts the induction hypothesis.

Note that the fact above implies that, at the end of step $k+1$, i.e., after the execution of PLANEXTENSION, we have that $u_{k+1} = \alpha_{k+1}$, since the algorithm moves in $Goals_{k+1}$ all the candidates with cost α_{k+1}, which is greater than the previous maximum cost u_k.

Now assume that the property to be proved is falsified at step $k+1$. This implies that there exist one or more states s s.t. $Cost(s) \leq u_{k+1}$ but $s \notin Goals_{k+1}$. Let us choose among these states the one with minimum cost. Since we know that $u_{k+1} = \alpha_{k+1}$, we can also write that $Cost(s) \leq \alpha_{k+1}$.

By induction hypothesis, since a state which is not in $Goals_{k+1}$ could not also be in $Goals_k$, we have that $u_k < Cost(s)$. Let us consider a cost-optimal strong plan for s. Such plan must contain at least one state $s' \notin Goals_k$. Indeed, if all the states of such plan were in $Goals_k$, then s should be in $Goals_{k+1}$. Let us choose among these states the one with minimum cost.

We have two cases:

- if $s' = s$, then we have that, for some suitable action a, $F(s,a) \subseteq Goals_k$. This would imply that $s \in Cand_{k+1}$ and, since $Cost(s) \leq \alpha_{k+1}$, we would have that $Cost(s) =$

α_{k+1} (recall that α_{k+1} is the minimum cost of a candidate in $Cand_{k+1}$). But in this case the algorithm would move s in $Goals_{k+1}$, contradicting the hypothesis;

- if $s \neq s'$, then $Cost(s') < Cost(s)$ (by definition of cost of a plan). Again, since $Cost(s) \leq \alpha_{k+1}$, we have that $Cost(s') < \alpha_{k+1}$, so $s' \notin Cand_{k+1}$. Thus we also have that $s' \notin Goals_{k+1}$, and this contradicts the hypothesis since s would not be the state with minimum cost s.t. $Cost(s) \leq \alpha_{k+1}$ and $s \notin Goals_{k+1}$.

□

Thus, if a state enters in the extended goals (and is therefore included in the strong optimal plan), then its cost, i.e., the cost of the corresponding strong plan, cannot be improved. This shows the algorithm correctness.

The algorithm completeness can be easily derived from Proposition 5, too. To this aim, we can use the following proposition.

Proposition 6 (Completeness). *Let $s \in S$. If s has a cost-optimal strong plan P, whose cost is not greater than $Cost(s_0)$, then there exists $k > 0$ s.t. $s \in ExtGoals_k$ and $(s, a, F(s, a))$ is added to SP.*

Proof. The proof follows from Proposition 5. Indeed, we have that the minimum cost of a candidate $\alpha_{k'}$ is strictly increasing in each step of the algorithm (otherwise, $u_{k'} < \alpha_{k'+1}$ would not hold). Thus, the process will eventually end with one of the following conditions:

- the initial state s_0 is in $ExtGoals_k$ (if a strong plan exists for such state): in this case, at step k all the states whose cost is not greater than $Cost(s_0)$, including s, are guaranteed to be in $ExtGoals_k$, too.

- there are no more candidate states that can be reached from the (extended) goals: in this case, since by hypothesis s has a strong plan, thus it can reach the goal, it would be included in the last set $ExtGoals_k$.

□

Finally, the algorithm termination is guaranteed by the arguments used in Proposition 6. Indeed, since the minimum cost of a candidate is strictly increasing, the algorithm will eventually build the cost-optimal strong plans for the states with highest cost: at this point, no new candidates will be available, and the process will terminate.

11.8 SUPMurphi: The Strong Algorithm Implementation into V-UPMurphi

In this section we describe how to model non-deterministic domains and how to synthesise the Strong Plan (if any) applying the Cost-Optimal Strong Planning Algorithm (see Procedure 16).

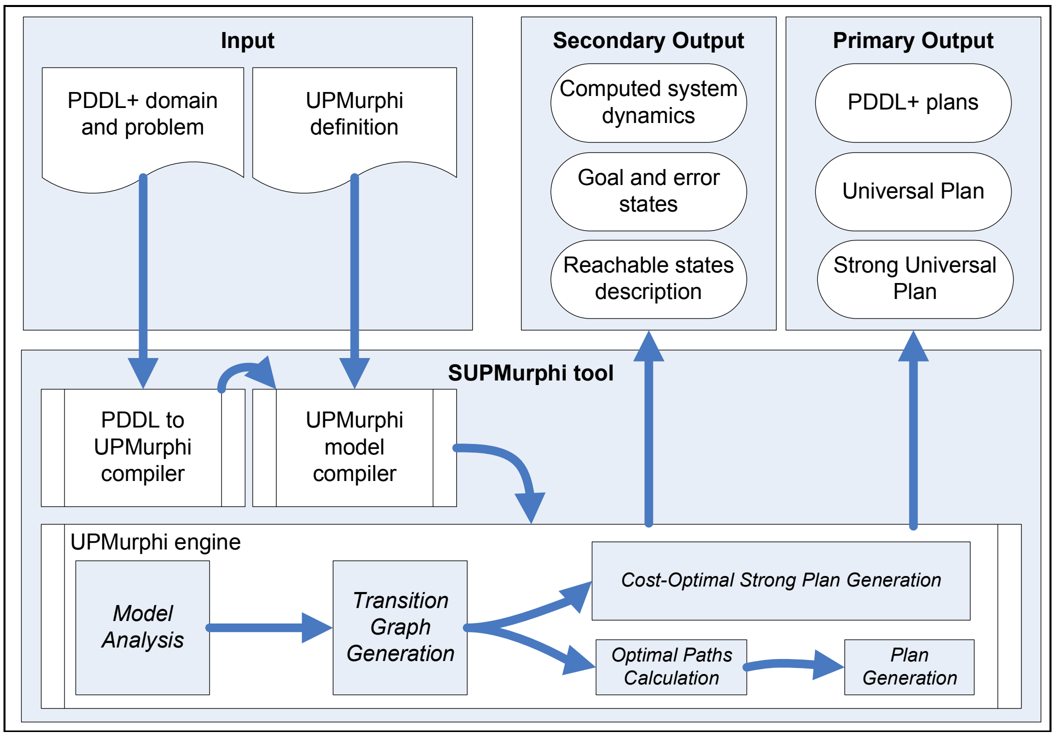


Figure 11.2: Overall structure of the SUPMurphi tool

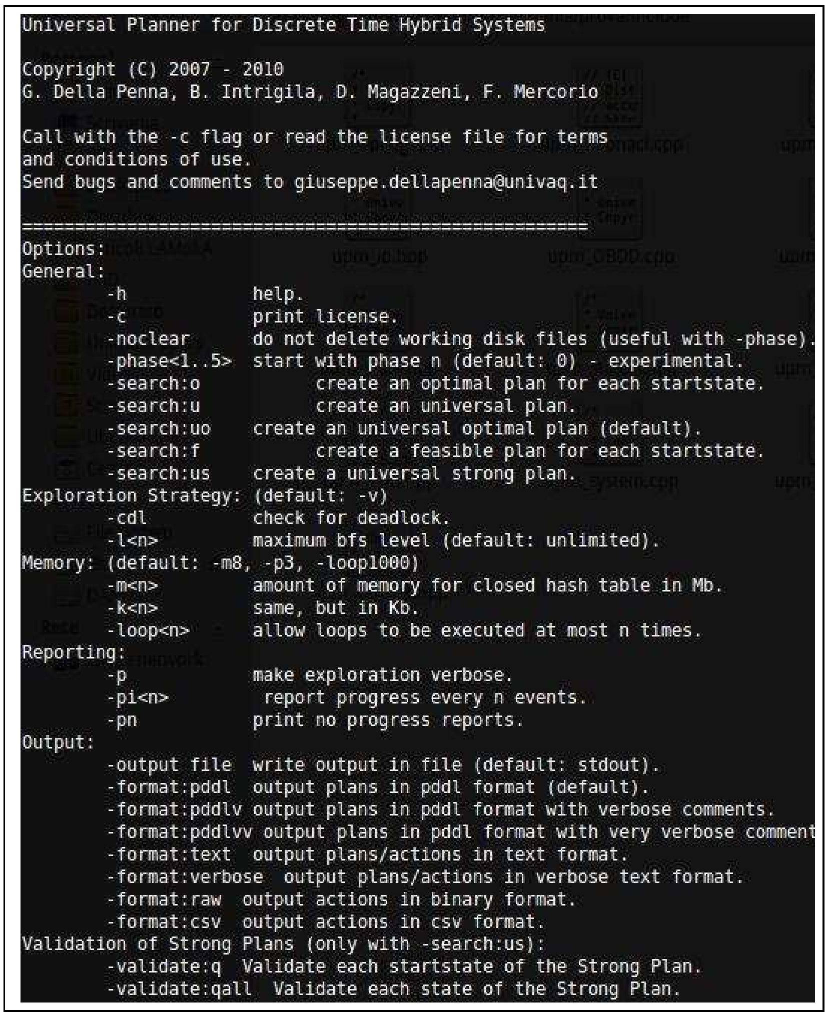
Figure 11.2 shows the new SUPMurphi overall structure in which:

1. The UPMurphi core provides the ability to manage PDDL+ domains and to explore the dynamics of the domain, as described in Section 6.5.

2. The disk-based algorithm (described in Chapter 7) allows one to exploit the use of the disk during all phases (i.e., the strong algorithm can work directly on disk data structures described in Section 7.2.1). As discussed in Chapter 11, the strong algorithm requires that the *Pre(s)* function is given, in order to access to a *predecessor* of a node in linear time (with respect to the number of outgoing edges of a given node). To this aim, the *Transition Graph Generation* phase has been adapted to store the graph dynamics in both directions (i.e., directed and inverted form), creating two *Transition Graph* files.

   Clearly SUPMurphi still maintains the capability to adapt these files to the system size, choosing *automatically* among three different modalities: **Memory Mode** when the size of the two graphs fits into the RAM, **Mixed Mode** if only one graph can be stored into RAM and the other one on the disk, and **Disk Mode** if both graphs are stored on disk.

3. Implements the the Cost-Optimal Strong Planning Algorithm detailed above. Figure 11.3 is a snapshot of the new options helper in which `-search:us` enables the synthesis of strong plans after the transition graph generation. Then, if a strong plan exists options `-validate:q` and `-validate:qall` verifies the generated solution, starting from each startstate and each strong state respectively.



```
Universal Planner for Discrete Time Hybrid Systems

Copyright (C) 2007 - 2010
G. Della Penna, B. Intrigila, D. Magazzeni, F. Mercorio

Call with the -c flag or read the license file for terms
and conditions of use.
Send bugs and comments to giuseppe.dellapenna@univaq.it

======================================================
Options:
General:
        -h              help.
        -c              print license.
        -noclear        do not delete working disk files (useful with -phase).
        -phase<1..5>    start with phase n (default: 0) - experimental.
        -search:o             create an optimal plan for each startstate.
        -search:u             create an universal plan.
        -search:uo      create an universal optimal plan (default).
        -search:f             create a feasible plan for each startstate.
        -search:us      create a universal strong plan.
Exploration Strategy: (default: -v)
        -cdl            check for deadlock.
        -l<n>           maximum bfs level (default: unlimited).
Memory: (default: -m8, -p3, -loop1000)
        -m<n>           amount of memory for closed hash table in Mb.
        -k<n>           same, but in Kb.
        -loop<n>        allow loops to be executed at most n times.
Reporting:
        -p              make exploration verbose.
        -pi<n>           report progress every n events.
        -pn             print no progress reports.
Output:
        -output file  write output in file (default: stdout).
        -format:pddl  output plans in pddl format (default).
        -format:pddlv output plans in pddl format with verbose comments.
        -format:pddlvv output plans in pddl format with very verbose comment
        -format:text  output plans/actions in text format.
        -format:verbose  output plans/actions in verbose text format.
        -format:raw  output actions in binary format.
        -format:csv  output actions in csv format.
Validation of Strong Plans (only with -search:us):
        -validate:q  Validate each startstate of the Strong Plan.
        -validate:qall  Validate each state of the Strong Plan.
```

Figure 11.3: A snapshot of the SUPMurphi helper

To clarify how a non-deterministic domain can be modelled, we use the *Hurried Passenger Problem* as introduced in Section 11.4. The SUPMurphi model is given in Figure 11.4. The keyword `ruleset` is used to model non-determinism of an action whilst the keyword `weight` models the cost of the transitions, which can be a parametric function.
In our example, the state is composed by the location of the passenger and the local clock time. The `startstate` construct models the initial position and clock time of the passenger (i.e., the 6*am* at *home*). Then, each flight of Table 11.1 is modelled by an action that

requires to stay in the airport at least one hour before the departure time. The cost of a flight is given by function `w()` which sums the time spent in the airport waiting for the flight, the duration of the flight and the delay of the flight (which could be zero).

Figure 11.7 shows a complete SUPMurphi log execution in which the V-UPMurphi phases are performed (i.e., the *Model Analysis*, and the *Transition Graph Generation*). Then, the phase *Cost-Optimal Strong Plan* phase starts and synthesises the strong plan in 5 steps. The solution is verified (i.e., the tool verifies if each strong plan always reaches a goal). Finally, Figure 11.6 shows the solution stored in the `hurried.strong` file. For the sake of completeness, the graph of reachable states generated by UPMurphi is reported in Figure 11.5.

It is worth noting that the solution is slightly different from the one given in Figure 11.2, that is the tour *Ciampino-Amsterdam-San Francisco* is impracticable due to the late arrival time in *CIA* airport.

```
const
 HOME: 0; FCO : 1; CIA : 2; CDG : 3;
 BER : 4; AMS : 5; SFO : 6; GMT7: 7;
type
 day_type : 0..24;
 delay_t : 0..1;
 location_type: 0..6;
 start_type: 6..6;
var
 time[pddlname: time;]: day_type;
 location[pddlname: location;]:
  location_type;
-- the weight is given by time to wait in
  airport + flight time + delay
function w(delay: delay_t ; departure:
 day_type; length: day_type) : day_type;
begin
 return ((departure-time)+(length+delay))
  ;
end;

ruleset t: start_type do
 startstate "At Home"
 time := t;
 location := HOME;
 end;
end;

rule "Q" (location=HOME & time<8)==>
weight: 1;
 begin
  location:=FCO;
  time:= time + 1;
 end;

rule "P" (location=HOME & time<5)==>
weight: 1;
 begin
  location:=CIA;
  time:= time + 1;
 end;

ruleset delay : delay_t do
rule "FlightA" (location=FCO & time<8)==>
weight:  w(delay,8,1);
 begin
  location:=CDG;
  time:=time+(8-time)+(1+delay);
 end; end;

ruleset delay : delay_t do
rule "FlightB" (location=CDG & time<10)
 ==>

weight: w(delay,10,9);
 begin
  location:=SFO;
  time:=time+(10-time)+(9+delay)-GMT7;
 end; end;
```

```
ruleset delay : delay_t do
rule "FlightC" (location=CDG & time<19)
 ==>
weight: w(delay,19,9);
 begin
  location:=SFO;
  time:=time+(19-time)+(9+delay)-GMT7;
 end; end;
ruleset delay : delay_t do
rule "FlightD" (location=CIA & time<5)==>
weight: w(delay,5,3);
 begin
  location:=AMS;
  time:=time+(5-time)+(3+delay);
 end; end;

ruleset delay : delay_t do
rule "FlighE" (location=FCO & time<8)==>
weight: w(delay,8,2);
 begin
  location:=BER;
  time:=time+(8-time)+(2+delay);
 end; end;

ruleset delay : delay_t do
rule "FlightF" (location=BER & time<11)
 ==>
weight: w(delay,11,10);
 begin
  location:=SFO;
  time:=time+(11-time)+(10+delay)-GMT7;
 end; end;

ruleset delay : delay_t do
rule "FlightG" (location=BER & time<12)
 ==>
weight: w(delay,12,10);
 begin
  location:=SFO;
  time:=time+(12-time)+(10+delay)-GMT7;
 end; end;

ruleset delay : delay_t do
rule "FlightH" (location=AMS & time<15)
 ==>
weight: w(delay,15,10);
 begin
  location:=SFO;
  time:=time+(15-time)+(10+delay)-GMT7;
 end; end;

ruleset delay : delay_t do
rule "FlightI" (location=BER & time<12)
 ==>
weight: w(delay,12,1);
 begin
  location:=AMS;
  time:=time+(12-time)+(1+delay);
 end; end;

goal "On Time"
   (location=SFO & time <= 21);
metric: minimize;
```

Figure 11.4: The *Hurried Passenger Problem* model as described in Section 11.4, page 127.

```
-- Source: Action(cost)->Target
State 0: Q(1)->1
State 1: FlightA(3)->5 FlightA(2)->4 FlightE(4)->3 FlightE(3)->2
State 2: FlightF(12)->8 FlightF(11)->10 FlightG(13)->9 FlightG(12)->8
  FlightI(4)->7 FlightI(3)->6
State 3: FlightG(12)->14 FlightG(11)->13 FlightI(3)->12 FlightI(2)->11
State 4: FlightB(11)->18 FlightB(10)->17 FlightC(20)->16 FlightC(19)->15
State 5: FlightC(19)->20 FlightC(18)->19
State 6: FlightH(13)->22 FlightH(12)->21
State 7: FlightH(12)->24 FlightH(11)->23
State 11: FlightH(13)->22 FlightH(12)->21
State 12: FlightH(12)->24 FlightH(11)->23
```

Figure 11.5: The *Hurried Passenger Problem* graph of model in Figure 11.4

```
 -- Strong Plan Filename: hurried.strong(text mode)
 -- (source[type],action name,Max Cost to goal) --> (reached states list)
 -- [I] = startstate [G] = goalstate [IG] = both start and goal state
(0[I],Q,17)-->(1)
(1,FlightE,16)-->(3, 2)
(2,FlightF,12)-->(8[G], 10[G])
(3,FlightG,12)-->(14[G], 13[G])
(4,FlightB,11)-->(18[G], 17[G])
(6,FlightH,13)-->(22[G], 21[G])
(7,FlightH,12)-->(24[G], 23[G])
(11,FlightH,13)-->(22[G], 21[G])
(12,FlightH,12)-->(24[G], 23[G])
```

Figure 11.6: SUPMurphi strong plan for The *Hurried Passenger Problem* model of Figure 11.4

```
Launch: hurried -search:us -validate:qall

=== Analyzing model... ==============================
Model exploration complete (in 0.10 seconds).
 29 rules fired
 1 start states
 25 reachable states
 13 goals found
=== Building model dynamics... ======================
* Transition Graph mode: Memory Image
* Maximum size of graph: 606060 transitions.

 Model dynamics rebuilding complete (in 0.10 seconds).
 25 states
 29 transitions
 out degree: min 0 max 2 avg 1.16

=== Looking for Strong Plans... ======================
Strong Plan Algorithm is going to run on a Graph having:
 25 States
 1 Start States
 13 Goal States
 29 Transitions
 11 Nondeterministic Actions
 20 Nondeterminism Degree

[0:0:0.10] Step: 0, Candidates: 0, ExtGoals: 13,
  OldExtGoals: 0, StrongPlan Size: 0
[0:0:0.10] Step: 1, Candidates: 6, ExtGoals: 14,
  OldExtGoals: 13, StrongPlan Size: 1
[0:0:0.10] Step: 2, Candidates: 2, ExtGoals: 18,
  OldExtGoals: 14, StrongPlan Size: 5
[0:0:0.10] Step: 3, Candidates: 1, ExtGoals: 20,
  OldExtGoals: 18, StrongPlan Size: 7
[0:0:0.10] Step: 4, Candidates: 0, ExtGoals: 21,
  OldExtGoals: 20, StrongPlan Size: 8
[0:0:0.10] Step: 5, Candidates: 0, ExtGoals: 22,
  OldExtGoals: 21, StrongPlan Size: 9

====================================================
Enjoy: Strong Plan found
 Strong Plan algorithm complete (in 0.10 seconds).
 6 steps done
 1 start states
 13 Goal states
 Strong Plan size: 9
 Strong Plan maxmimum cost: 17
 Strong Plan Filename: hurried.strong(text mode)
=== Validation of Strong Plans... ======================
 Strong Plan Validation complete (in 0.10 seconds).
 Processed: 9,
 Strong Startstates: 1,
 Not Strong Startstates: 0,
 Strong States: 8,
 Not Strong States: 0
```

Figure 11.7: SUPMurphi execution for The *Hurried Passenger Problem* model of Figure 11.4

CHAPTER 12

STRONG PLANNING CASE STUDIES

In this chapter we show two case studies for which the Cost-Optimal Strong Plan Algorithm of Chapter 11.5 has been applied.

The former (namely, the *inverted pendulum on a cart*) is aimed to show a real-world problem in which a cost-optimal strong plan exists. We provide the SUPMurphi model and some experimental results about the robustness of the strong solution (i.e., how the non-determinism affects the existence of the strong plan).

On the contrary, the second case study is inspired by the field of construction industry and it represents an industrial experience in which we used SUPMurphi in two respects: (1) we analysed the system dynamics of a proprietary system, and (2) we tried to synthesise a strong solution for it. Unfortunately, no strong plan could be devised in this case, since the problem does not allow a strong solution, as proved by the model checking based analysis.

12.1 THE INVERTED PENDULUM ON A CART

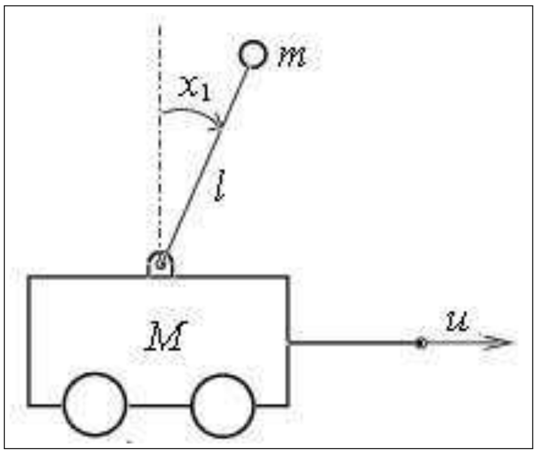


Figure 12.1: Inverted pendulum on a cart.

The *inverted pendulum on a cart* (depicted in Figure 12.1) is a hybrid system in which an optimal universal plan has to balance the pole in the vertical position by applying an appropriate horizontal force to the cart.

Note that this apparently simple case study is instead an important issue in the controller design for many real-world systems. Indeed, many control problems, e.g., engineering (i.e., the regulation of a steering antenna [104]) or robotics [169] can be reduced to an inverted pendulum problem. Indeed, previous works dealing with this system are based on neural network [7], as well as cell mapping [141, 153] to minimise the *time* spent to reach the equilibrium [153].

12.1.1 System Modelling

The system is described as presented by Papa et. al. in [141]. The pendulum state is described by two real variables:

- x_1 is the pendulum angle (w.r.t. the vertical axis) with $x_1 \in [-1.5, 1.5]$ rad with steps of $0.001 rad$;
- x_2 is the angular velocity with $x_2 \in [-8, 8]$ rad/sec with steps of $0.01 rad/sed$.

The continuous dynamics is described by a system of differential equations:

$$\begin{cases} \dot{x_1} = x_2 \\ \dot{x_2} = \frac{g_e \sin(x_1) - [\frac{\cos(x_1)}{m_p + m_c}][m_p l x_2^2 \sin(x_1) + u]}{4l/3 - [m_p/(m_p+m_c)] l \cos^2(x_1)} \end{cases}$$

where g_e is the gravitational constant, $m_p = 0.1 Kg$ is the mass of the pole, $m_c = 0.9 Kg$ is the mass of the cart, $l = 0.5m$ is the half-length of the pole and $u \in [-50, -46, \dots, 46, 50]$N is the force applied to the cart.

The actions that can be applied in each state of the system correspond to the force applied to the cart, i.e., $\mathcal{A} = [-50, -46, \dots, 46, 50]$. In this setting, the *cost of a transition* is given by the absolute value of the applied force, i.e., for any s, s', $\mathcal{W}(s, a, s') = |a|$. Therefore, a plan of minimum cost minimises the *worst-case* energy consumption. The time is sampled with a precision of 0.01 seconds.

The *non-determinism* of the system is given by possible disturbances on the actuator that may result in a small variation of the force actually applied. Hence, due these disturbances, x_2 can non-deterministically assume, with uniform probability distribution, a value that differs from the expected one by a small $\lambda \in [-\Lambda, \Lambda]$ with steps of 0.01 rad/sec.

To this aim we exploited the strong algorithm described in Section 11.5 to find a strong solution (if any). In such context, a strong universal plan is a controller composed by a tuple $< s, a, c >$. More precisely:

1. the s, a pair describes the actual position of the pendulum (i.e., the values of x_1 and x_2) and it suggests to apply the non-deterministic action a.
2. the c is the maximum Newton force that might be required (in the worst-case) to bring the pendulum from s to the equilibrium state via action a.
3. the action a is guaranteed to be the best choice between all the possible actions available which lead to a goal state (or to another controlled state).

```
const
 MAX_THETA : 1500; --in 0.001 rad
 MIN_THETA: -1500; --in 0.001
 MAX_THETA_DOT : 800; --in 0.01 rad/sec
 MIN_THETA_DOT :-800; --in 0.01 rad/sec
 MIN_U : -50;
 MAX_U : 50;

 TOLL_THETA : 15;
 TOLL_THETA_DOT : 7;
 L : 0.5; --Half-length
 M : 0.1; --Mass
 MAX_START_STEPS : 50 ;

type
 real_type : real(4, 99);
 theta_type : MIN_THETA..MAX_THETA;
 theta_dot_type : MIN_THETA_DOT..MAX_THETA_DOT;
 start_steps : 0..MAX_START_STEPS-1;
 dist_type : -K..K; -- model the non-determinism

var
 theta: theta_type;
 theta_dot: theta_dot_type;
 failure: boolean;

externfun next_theta (theta : theta_type; theta_dot : theta_dot_type) : theta_type
  "CGMURPHI_IP_controller_library.h";
externfun next_theta_dot (theta : theta_type; theta_dot : theta_dot_type; dist_L :
  real_type; dist_M : real_type; u:u_type ; dist:dist_type) : theta_dot_type;

function Equilibrium(theta : theta_type; theta_dot : theta_dot_type) : boolean;
begin
 return (theta <= 0.0 + TOLL_THETA & theta >= 0.0 - TOLL_THETA &
   theta_dot <= 0.0 + TOLL_THETA_DOT & theta_dot >= 0.0 - TOLL_THETA_DOT);
end;

function InRange(theta : ext_theta_type; theta_dot : ext_theta_dot_type) : boolean
 ;
begin
 return (theta <= MAX_THETA & theta >= MIN_THETA &
   theta_dot <= MAX_THETA_DOT & theta_dot >= MIN_THETA_DOT );
end;
```

Figure 12.2: The *inverted pendulum on a cart* model (first part)

```
 ruleset tmp_theta : start_steps do
  ruleset tmp_theta_dot : start_steps do
   startstate "source_state"
     theta := MIN_THETA + tmp_theta*30 ;
     theta_dot := MIN_THETA_DOT + tmp_theta_dot*16;
    failure := false;
   end;
  end;
 end;

ruleset dist : dist_type do
rule "Apply Force 0 N" (!( Equilibrium(theta, theta_dot))) ==>
weight: MAX_U - 0*4;
  var tmp_theta: ext_theta_type;
   tmp_theta_dot: ext_theta_dot_type;
   tmp_u : int_type;
 begin
  tmp_u := MIN_U + 0*4 ;
  tmp_theta := next_theta(theta, theta_dot) ;
  tmp_theta_dot := next_theta_dot(theta, theta_dot, L, M, tmp_u,dist) ;

  if (InRange(tmp_theta,tmp_theta_dot)) then
   theta := tmp_theta;
   theta_dot := tmp_theta_dot;
  else failure:= true;
  endif;
 end;
end;
....
....
ruleset dist : dist_type do
rule "Apply Force 25 N" (!( Equilibrium(theta, theta_dot))) ==>
weight: MIN_U + 25*4;
  var tmp_theta: ext_theta_type;
   tmp_theta_dot: ext_theta_dot_type;
   tmp_u : int_type;
 begin
  tmp_u := MIN_U + 25*4 ;
  tmp_theta := next_theta(theta, theta_dot) ;
  tmp_theta_dot := next_theta_dot(theta, theta_dot, L, M, tmp_u,dist) ;

  if (InRange(tmp_theta,tmp_theta_dot)) then
   theta := tmp_theta;
   theta_dot := tmp_theta_dot;
  else failure:= true;
  endif;
 end;
end;
```

Figure 12.3: The *inverted pendulum on a cart* model (second part)

12.1.2 Strong Universal Plan

We generated a start states cloud of 2500 defined as the set $\{(x_1,x_2)|x_1 \in [-1.5,\ldots,-0.03] \wedge x_2 \in [-8,-0.16]\}$ with steps of $0.03rad$ on x_1 and $0.16rad/sec$ on x_2. Roughly speaking, the set of start states represent all the positions (and angular velocity) in which the pendulum can be in a range of about 90 degrees.

Then, we considered different instances of the problem, taking into account disturbances of increasing size, i.e., with $\Lambda \in \{0.01, 0.02, 0.03, 0.04\}$. For each instance, we applied the strong algorithm on the sample set of initial states, which produced the results summarised in Table 12.1. Here, for each problem instance, we report some statistics about the corresponding graph *G*, i.e., total number of states ($|S|$), the number of reachable states and edges (*Reach* and *Reach*$_\tau$, respectively), the number of actions ($|\mathcal{A}|$) and the average and maximum out degree ($avg(\delta(s))$ and $\max(\delta(s))$, respectively) of the states.

To perform the synthesis of strong plans, we exploited the use of the disk both during the *Model Analysis* (as implemented in V-UPMurphi) and the *Cost Optimal Strong Plan* phase, which required to use disk for the third and the fourth instances.

Then, we summarise the corresponding cost-optimal strong plan *SP*, as devised by the algorithm, giving its size (i.e., the number of plans that can be extracted from *SP*), the maximum and minimum cost ($\max(\mathcal{C}(s_0)$ and $\min(\mathcal{C}(s_0))$ of a strong plan starting from an initial state (i.e., the minimum and maximum amount of energy required to reach a goal from a startstate in the worst-case). It is worth noting that the maximum amount of energy required could be high (more than $3,000N$ for the first instance). Nevertheless, this value refers to a plan which requires to apply 135 actions to reach a goal.

Therefore we may note that, as expected, the greater the size of disturbances, the bigger the number of transitions, the smaller the number of states for which a strong plan is found, that are about 53% for the first instance and 38% for the third one, whilst for the fourth instance, no strong plan exists.

The synthesis time (which includes all the phases) never required more than one hour, using a Linux machine equipped with an Intel x86 CPU at 2.66Ghz, with $3Gb$ of RAM for the hash table.

Finally, for the sake of completeness, we have to note that the results shown in Table 12.1 differ from the ones presented in [53] (with respect to instances and synthesis time). This is mainly due to the implementation of the SUPMurphi tool, which uses the V-UPMurphi algorithm and data structures to efficiently implement the Strong Planning Algorithm. Thanks to SUPMurphi, we were able to test the algorithm on (more) big instances of inverted pendulum.

Table 12.1: Experimental results for the inverted pendulum on a cart problem.

Instance		1	2	3	4
Λ		0.01	0.02	0.03	0.04
Graph	$\lvert S\rvert$	$5 \cdot 10^7$			
	Reach	$1,513,454$	$1,526,446$	$1,537,842$	$1,547,493$
	$Reach_\tau$	$98,728,221$	$172,743,090$	$233,520,157$	$301,773,021$
	$\lvert\mathcal{A}\rvert$	26	26	26	26
	$\max(\delta(s))$	78	130	182	234
	$\text{avg}(\delta(s))$	65.23	108.59	151.5	195.1
Disk	.transitions	800Mb	1.33Gb	1.87Gb	2.4Gb
	.graph (directed)	memory mode	memory mode	memory mode	2.4Gb
	.graph (inverse)	memory mode	memory mode	1.86Gb	2.4Gb
SP	size	$799,966$	$802,419$	$588,460$	—
	$\max(\mathcal{C}(s_0))$	$3,220N$	$3,178N$	$2,628N$	—
	$\min(\mathcal{C}(s_0))$	$38N$	$40N$	$46N$	—
	max plan length	131	128	95	—
	min plan length	3	4	4	—
	Iterations	$1,640$	$1,696$	$1,330$	1
	time (min)	17	25	52	37

12.2 The Building Lifting System

In this section we describe an industrial experience where SUPMurphi was applied, in the field of construction industry, to analyse and automatise a proprietary *building lifting system* showing that a strong plan, in some real world problems, may not exist. Nevertheless, formal methods can provide a valuable help in reducing human effort. In particular, we worked for a company specialised in the strengthening of deep foundations, consolidation, lifting and seismic isolation of buildings as well as in the construction of new basements under existing buildings. To this aim, the company developed proprietary hydraulic cylinders which can lift a building up to two meters above the ground. When the structure is raised, the foundations are strengthened with a seismic platform and a shock absorbing system.

12.2.1 The Lifting Process

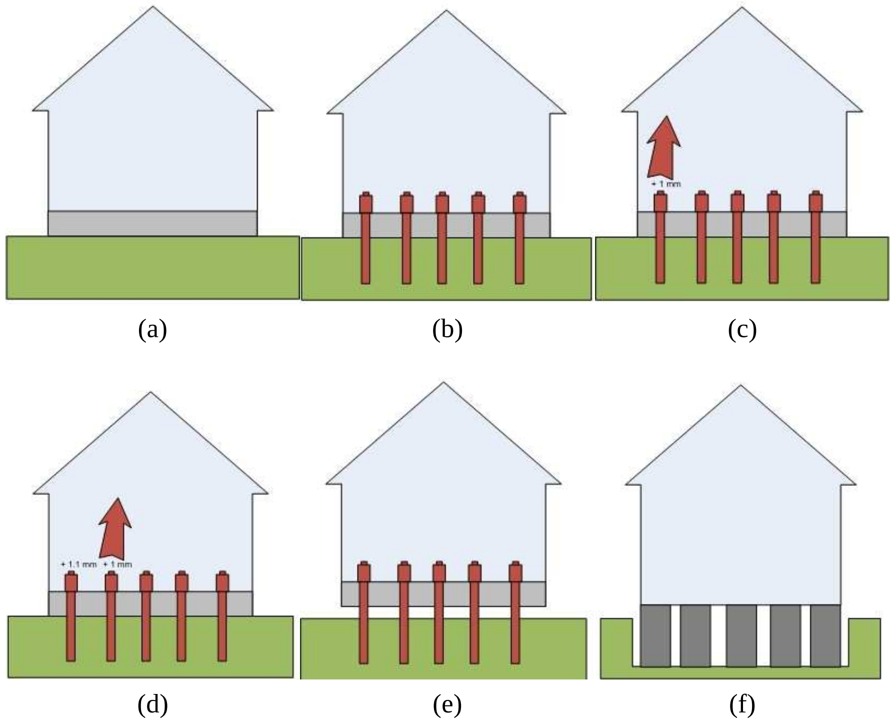


Figure 12.4: Illustration of a complete *lifting process* on a building.

Before starting the lifting process, construction engineers build a reinforced concrete structure that covers the base of the building (Figure 12.4a). Then, following a static analysis of the building, cylinders are suitably positioned on this structure up to two meters below the terrain level (Figure 12.4b) and connected to a constant-flow hydraulic pump through a pipeline. Displacement and pressure sensors (with a precision of $\frac{1}{10}mm$

and $\frac{1}{10}bar$, respectively) are placed on each cylinder to track the lifting process, which is governed by an operator through a graphical control interface and takes place as follows:

1. The operator defines a *lifting sequence*, i.e., a sequence of cylinders to activate.
2. The system executes the defined sequence by sending a hydraulic impulse to each cylinder through a junction box.
3. Each impulse produces a displacement of the activated cylinder (Figures 12.4c and 12.4d), and it may cause a partial displacement of other cylinders in the neighbourhood of the activated one.
4. If the displacement of all the cylinders is within a safe threshold, the sequence continues with the next cylinder, otherwise a manual intervention of the operator is required to return the system in a safe state.
5. The process continues iteratively with new lifting sequences until the building reaches the desired height (Figure 12.4e).

At the end of this process, the shock absorbing systems are placed below the structure (Figure 12.4f).

Note that the lifting process is very slow, i.e., about 1 cm/h (where the target elevation is on average between 1 and 2 meters). Indeed, the execution of a single lifting sequence can raise the building up to 1 *mm* and, since the sequence is manually defined by the operator, there is a high probability to make *selection errors* (i.e., activations of the wrong cylinder), which may lead to the violation of safety or structural constraints, or to *hydraulic blocks* (i.e., the pressure of a cylinder exceeds the safe limit). In these cases, the operator must often perform manual adjustments, wait for the system to return in an acceptable state, and then decide an alternative activation sequence, further delaying the lifting process. Therefore, to speed up the process, we exploited the disk-based UPMurphi described in Section 7.2 to analyse it and then try to introduce more automatisation where it was possible.

12.2.2 System Modelling

We started our analysis by tracking a complete lifting process in a construction site, and then evaluating the logs produced during other successful lifting processes, which report the activation sequences used and the corresponding cylinders behaviour during the entire process. These observations allowed us to extract some important domain properties and constraints, which were then used in the next phase in order to model a realistic and accurate model. First of all, we formalised the concept of activation sequence as follows.

Definition 24 (Lifting Process and Activation Sequence)**.** *Let H be the set of cylinders. A* lifting process *P is defined as a sequence of activation sequences $P = \left(S^H{}_1, \ldots, S^H{}_n\right)$ with $n \geq 1$. An* activation sequence *is a sequence $S^H = (h_1, \ldots, h_m)$ with $h_i \in H$ and $m \leq |H|$.*

In our experiments the number of cylinders ($|H|$) was fixed to seven. The actual number of cylinders used in each lifting process may vary w.r.t. the building area, however seven represents a reasonable average number. Moreover, we noted that the entire system is reset after each activation sequence (e.g., the cylinders pressure is stabilised, and their current displacement becomes the new "zero height"), in a way that makes each sequence independent from the others. Thus, our study focused on the analysis and the automatisation of *single activation sequences*, rather than on the entire lifting process.

Each cylinder h has an associated *displacement*, (i.e., its current height), indicated by $dsp(h) \in [0, 0.1, \ldots, 1.2]mm$. Each element in an activation sequence indicates the activation of the corresponding cylinder, which modifies its displacement and may also affect its *neighbours*, defined as follows.

Definition 25 (Cylinder Neighbours)**.** *Let $d_H(h,h')$ be the Euclidean distance between cylinders $h,h' \in H$, and $d(h,h')$ the corresponding normalised distance obtained as $d(h,h') = \frac{d_H(h,h')}{\max_{\bar{h},\bar{h}' \in H}(d_H(\bar{h},\bar{h}'))}$. Then $h,h' \in H$ are* neighbours *if $d_H(h,h') < \delta$. We denote with $ngh(h)$ the set of all neighbors of h.*

The value of δ depends on the placement of the cylinders and from mechanic and elastic characteristics of the building basement, and it is estimated by the construction engineers. Intuitively, two cylinders are neighbours if they are "close enough" to make the selection of the first also affect the displacement of the second.

Thus, the activation of a cylinder h modifies its displacement and possibly induces a displacement variation (*induced displacement*) to all the neighbour cylinders $h' \in ngh(h)$. However, neither the displacement nor the induced displacements can be expressed by a deterministic formula. Rather, they can assume *non-deterministically* any value from a set of possible results (which were empirically identified during our preliminary study), with uniform probability.

Definition 26 (Displacement and Induced Displacement)**.** *Let h be a cylinder and $h' \in ngh(h)$ a neighbour cylinder for h, with $dsp(h) = d_h$ and $dsp(h') = d_{h'}$. After the activation of h, the new displacements $dsp(h) = d'_h$ and $dsp(h') = d'_{h'}$, respectively, always satisfy the following constraints: $|d_h - d'_h| \in [0.8, \ldots, 1.2]$, $d'_{h'} = d_{h'} + idsp(h,h')$, where $idsp(h,h') = x \cdot \frac{1}{d(h,h')}$, $x \in [0.0, \ldots, 0.6]$ is called the* induced displacement.

Finally, the domain analysis evidenced some constraints, which are critical for the correct execution of the lifting process.

C1 : At any point of the lifting process, the following must hold: $\forall h,h' \in H$, $|dsp(h) - dsp(h')| \leq 1$mm.

C2 : Let $S^H = (h_1, \dots, h_m)$ be an activation sequence, then $\forall i \in 1 \dots m-1$, $h_i \neq h_{i+1}$.

C3 : Let $S^H = (h_1, \dots, h_m)$ be an activation sequence. Any cylinder h which, at a given step $i \in 1 \dots m$, reaches its final displacement, i.e., $1.0 \leq dsp(h) \leq 1.2$, must not be further activated.

Constraint **C1** is a *safety constraint* ensuring that the building does not collapse. Constraint **C2** forbids consecutive selections of the same cylinder (which may apply too much pressure to a single part of the building) and the constraint **C3** guarantees that, when a cylinder reaches its goal displacement (i.e., about 1 *mm*) it will not be further activated. Any cylinder activation whose effect violates at least one of the constraints above causes a selection error.

It is worth noting that we were not able to model the *pressure evolution* of cylinders. Indeed we observed that, during the lifting process, the pressure behaviour is subject to unpredictable environmental conditions (e.g., temperature, presence of concrete structures near the cylinder). On the other hand, we also empirically observed that hydraulic blocks are tightly linked to selection errors, thus avoiding selection errors we can reasonably prevent also hydraulic blocks.

12.2.3 System Analysis

After modelling the activation sequences as described in the previous section, we first used the ModelAnalysis procedure (see Procedure 5, page 59) to analyse their dynamics in order to build the corresponding transition graph.

Table 12.2: SUPMurphi Statistics

Graph		*Universal Plan*	
State Space Size	$3.2 \cdot 10^{16}$	Size (plain/OBDD)	71/3.4 MB
State Size (compressed/not)	12/20 bytes	Memory peak	1,245 MB
Reachable States	120,350,719	Plans	49,326,019
Transitions	184,445,662	Time (sec)	1,136.51
Transition File Size	2,692 MB		
Graph File Size	2,631 MB		

The analysis started from a single initial configuration where all cylinders displacements are zero. We specified as goal any state where all the cylinder displacements are between 1 and 1.2*mm*, and all the states corresponding to selection errors as error states. In this way, SUPMurphi was able to record all the success and failure states, as well as the control paths that lead to them. All the experiments were done on a 32 bits 2.2Ghz CPU equipped with 3 GB of RAM. The results are summarised in the *Graph* section of Table 12.2.

Given the domain variables and their ranges (induced by Definition 26), we can easily calculate the state space size of the system, which is about of 10^{16} states. Thanks to the reachability analysis, SUPMurphi was able to build a relatively smaller transition graph, with more than 120 million nodes and 180 million edges.

12.2.4 Strong Universal Plan

In our initial attempt, we tried to achieve a complete automatisation of the lifting process. To this aim, we are interested in the synthesis of a *strong* universal plan, i.e., a plan able to reach the right height never occurring in a selection error.

Hence, we used our strong algorithm to synthesise a cost optimal strong plan, as described in Section 11.1. However, our *Building Lifting System* does not admit a strong solution. In other word, for each activation sequence always exists a cylinder activation that may lead to a selection error. This outcome was partly expected, since the non-determinism of the system responses make selection errors unavoidable.

It is worth noting that, in such kind of domain, only a strong solution would be acceptable. Indeed weak or strong cyclic plans (if any) did not have an acceptable reliability level to automate the lifting process.

12.2.5 Safest Universal Plan

Therefore, we tried another possible solution, i.e., we used V-UPMurphi looking for the *safest* activation sequences, which have less probability to encounter a selection error during their execution. In other words, we focused on the synthesis of a universal plan that *reduces* the selection error probability.

In order to determine the probability distribution of the selection error, we started again from the system dynamics. In addition, we used V-UPMurphi's ComputeOptimalPath algorithm (see Procedure 7, page 61) to generate a (possibly non strong) universal plan from the system transition graph created in the previous phase. The results are summarised in the *Universal Plan* section of Table 12.2. The tool synthesised about 50 million plans (i.e., possible activation sequences) in about 20 minutes, requiring a memory peak of 1.2GB. The entire universal plan size was about 70MB, which decreased up to 7.4MB thanks to the OBDD compression algorithm implemented into V-UPMurphi.

Then, we applied a post processing algorithm to look at the states and transitions used by the universal plan. More precisely, in each state we counted the transitions that could lead to error states (previously recorded by V-UPMurphi), and used this information to recursively calculate the error probability of each state as in the following definition:

Definition 27. *Let s be a state, $\{a_1, \dots, a_n\}$ all the possible actions that can be performed* *in s and $F(s,a_i)$ the state reached by each of them, respectively. The* error probability *of s, written as $ep(s)$ is recursively defined as*

$$ep(s) = \begin{cases} 1 & \textit{if s is an error state} \\ \frac{\sum_{a_i} ep(F(s,a_i))}{n} & \textit{if } n > 0 \\ 0 & \textit{otherwise} \end{cases}$$

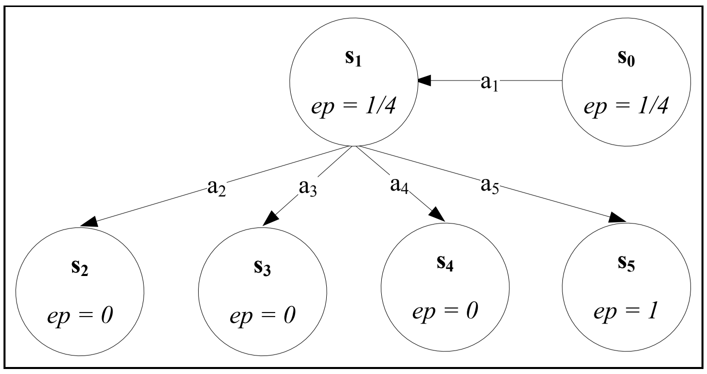


Figure 12.5: A fragment of transition graph with error probabilities

To give an example, Figure 12.5 contains a fragment of transition graph where the lower part of each node contains the corresponding error probability. In particular, the error probability of s_5 is 1, whilst it is zero for all the other states reachable from s_1. Thus, the error probability of s_1 is $1/4$ and the same is for s_0, since s_1 is its only child.

Having assigned an error probability to all the states, then the final safest universal plan will contain, for to each state, the plan that has the lower error probability.

To evaluate the accuracy of this universal plan we used a Monte Carlo based algorithm, performing a set of simulations exploiting the dynamics computed by V-UPMurphi during the first phase. The simulator works iteratively as follows: given the current state of the system, it looks in the universal plan for the action to take, applies it, chooses one of the possible (non-deterministic) outcomes of the action and makes the corresponding state the current one. The choice is made with uniform probability, since it was not possible to extract a more precise probability distribution from the domain data.

The simulation stops when the system reaches the goal or an error state. By running a large number of simulations and counting the success and failure results, we get an estimation of the universal plan accuracy, which shows that it is able to complete the lifting process only in 35% of the cases, whilst in the remaining 65% it must stop for an error.

12.2.6 DECISION SUPPORT SYSTEM

The build lifting system represents a real-case study which demonstrates that a strong plan (i.e., in this contexts a complete automatisation of the building lifting process) may not exist in the practice. Moreover, it is important to remark that, due to the high reliability requirements of the system, also weak or strong cyclic solutions (if any) did not have acceptable. Human surveillance and manual intervention will be always required to successfully complete it. However, all the artifacts produced during our study could be profitably exploited to create a *decision support system* which may lower the human efforts needed during the lifting process.

To this aim, we finally created a software tool which is designed to work “side by side” with the human operator. In particular, we extended the safest universal plan, making its application *interactive*. Indeed, in each step of the activation sequence, the tool is able to suggest to the user the best possible move to take, i.e., the cylinder activation which minimises the error probability. However, only if this probability is small enough, the action is performed automatically, otherwise the system asks for the operator approval. In either case, if the outcome is an error state or a state with a high error probability, the system triggers an alarm and notices the operator to manually perform a roll back action to return the system in an acceptable state before trying to activate another cylinder.

With this solution, we maintain the 35% of fully automatised successful lifting processes, whereas we can adjust the auto-activation probability threshold above to obtain a satisfactory compromise between the process safety and the need of operator choices which, however, are always “guided” by the suggested action.

In other words, thanks to the decision support system, the user is alerted before making any critical action (i.e., with a high error probability) which may put the system in an error state. Thus, the contribution of this artefact to the lifting process is mainly that it provides useful information in order to avoid system hangs and the consequent restore procedures, which usually require big efforts.

This last artefact, after being initially tested using a simulation scheme similar to the one described in the previous section, is now ready to be experimented on-field.

In the future, our idea is to put the decision support system side-by-side with the human operator during a lifting process, connecting it only to the displacement sensors: in this way, while the lifting is being carried on manually by the operator, we could look at the software and compare its decisions to the human ones.

CHAPTER 13

CONCLUSIONS

In this Thesis, we addressed the problem of dealing with systems having both discrete and continuous (possibly nonlinear) dynamics, and which may present a non-deterministic behaviour. We focused on the analysis and control of their dynamics, discussing how the problems of planning, universal planning, and strong planning have been handled in the literature.

To this regard, we showed that the model checking technique can be suitable to perform planning and universal planning for continuous systems, by analysing the dynamics of a Finite State System obtained from a Discrete Time Hybrid System, and generating optimal plans and controllers for it. As a contribution, we applied the explicit model checking technique, that by one side works well on systems having a continuous dynamics hard to invert, on the other side it is affected by the well-known state explosion problem. To mitigate this problem, we extended the UPMurphi tool by supporting the disk-based verification, which allows one to cope with systems having a large state space. Then, we tested this approach on a number of continuous case studies, many of which inspired by real world problems.

Moreover, we showed that the explicit model checking can be successfully applied to a different class of problems, that is the analysis and verification of data quality (i.e., consistency in our case) on sets of dirty dataset which can be modelled through FSS. To this aim, we defined a technique based on formal methods which increased the consistency of the data quality process. We modelled a real case scenario of a Public Administration Database, and we applied a modified version of CMurphi to test if such methodology improved the overall data quality process.

Finally, in the last part of this thesis, we discussed the problem to synthesise plans able to reach a goal in systems having a non-deterministic dynamics. We gave a survey on how this problem has been handled in the literature, an we provided a novel algorithm which synthesises cost-optimal strong plans, minimising the cost of the non-deterministic worst-case execution. We proved the correctness and completeness of the algorithm and we applied it on two real world problems.

All algorithms we presented in this thesis have been implemented in the following tools, all built on top of Murphi model checker:

V-UPMurphi: It is a computational engine to enhance the ability of UPMurphi (The Universal Planner Murphi) to synthesise plans and universal plans for (possibly non-linear) Discrete Time Hybrid Systems defined with a PDDL+ model. V-UPMurphi exploits the use of the disk during the exploration of the dynamics, as described in Chapter 7. It implements on disk the state space reduction techniques inherited from Murphi (as bit-compression, hash-compaction) and allows one to pause and resume the system analysis process, using the disk to both explore and store the graph of the system dynamics.

SUPMurphi: The Strong Universal Planner Murphi is built on-top of V-UPMurphi. In particular, it implements the cost-optimal strong planning algorithm, as described in Chapter 11. It uses the Murphi description language to model non-deterministic behaviour of systems' actions and, thanks to the use of disk-based algorithm, is able to perform strong planning. Then, it allows one to validate the final strong plan (if any) on the system graph.

13.1 FUTURE WORKS

Our future research activity is moving on two contexts.

In the context of *planning* and *control* problems, we are actually working to extend the application of the approach presented in this Thesis to deal with a wider class of systems, as well as to apply it on other real-world planning and control problems.
To this regard, we intend to exploit heuristics search during the system analysis as well as to enrich SUPMurphi with other state space reduction techniques.

In the context of *data quality*, currently we are further investigating the benefits that model checking can provide by applying both sensitivity analysis on dataset indicators and performing data cleansing through model checking.

REFERENCES

[1] International Conference on Automated Planning and Scheduling (ICAPS), url = `http://www.informatik.uni-trier.de/~ley/db/conf/aips/index.html`. (Cited on page 46.)

[2] IPC Web Page: `http://ipc.icaps-conference.org`, 2002. (Cited on page 14.)

[3] AFRATI, F. N., AND KOLAITIS, P. G. Repair checking in inconsistent Databases: Algorithms and Complexity. In *Proceedings of the 12th International Conference on Database Theory* (2009), ICDT '09, ACM, pp. 31–41. (Cited on page 99.)

[4] ALBORE, A., PALACIOS, H., AND GEFFNER, H. Compiling uncertainty away in non-deterministic conformant planning. In *ECAI* (2010), pp. 465–470. (Cited on page 118.)

[5] ALPERN, B., AND SCHNEIDER, F. B. Defining liveness. *Information Processing Letters 21*, 4 (1985), 181–185. (Cited on page 20.)

[6] AMINIFAR, F., FOTUHI-FIRUZABAD, M., KHODAEI, A., AND FARIED, S. Optimal placement of unified power flow controllers (UPFCs) using mixed-integer non-linear programming (MINLP) method. In *Power Energy Society General Meeting, 2009. PES '09. IEEE* (july 2009), pp. 1 –7. (Cited on page 36.)

[7] ANDERSON, C. W. Learning to control an inverted pendulum using neural networks. *IEE Control System Magazine*, 9 (1989), 31–37. (Cited on page 146.)

[8] ASARIN, E., MALER, O., AND PNUELI, A. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems* (1994), pp. 1–20. (Cited on page 2.)

[9] AYLETT, R., SOUTTER, J. K., PETLEY, G. J., AND CHUNG, P. W. H. AI planning in a chemical plant domain. In *Proc. ECAI 1998* (1998), pp. 622–626. (Cited on pages 3 and 47.)

[10] BAIER, C., AND KATOEN, J.-P. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. (Cited on pages xv, 1 and 18.)

[11] BARATEIRO, J., AND GALHARDAS, H. A Survey of Data Quality Tools. *Datenbank-Spektrum 14* (2005), 15–21. (Cited on page 98.)

[12] BATINI, C., CAPPIELLO, C., FRANCALANCI, C., AND MAURINO, A. Methodologies for Data Quality Assessment and Improvement. *ACM Comput. Surv. 41* (July 2009), 16:1–16:52. (Cited on page 100.)

[13] BATINI, C., AND SCANNAPIECO, M. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer, 2006. (Cited on pages 3, 97 and 98.)

[14] BEHRENDS, E. *Introduction to Markov Chains*. Vieweg, 2000. (Cited on page 124.)

[15] BEHRMANN, G., COUGNARD, A., DAVID, A., FLEURY, E., LARSEN, K. G., AND LIME, D. UPPAAL-TIGA: Time for playing games! In *19th International Conference on Computer Aided Verification (CAV)* (Berlin, Germany, July 3-7, 2007), W. Damm and H. Hermanns, Eds., vol. 4590 of *Lecture Notes in Computer Science*, Springer, pp. 121–125. (Cited on page 47.)

[16] BELL, K. R. W., COLES, A. J., COLES, A. I., FOX, M., AND LONG, D. The role of AI planning as a decision support tool in power substation management. *AI Communications 22*, 1 (2009), 37–57. (Cited on pages 3 and 47.)

[17] BERTOLI, P., CIMATTI, A., ROVERI, M., AND TRAVERSO, P. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proc. 17th IJCAI* (2001), Morgan Kaufmann, pp. 473–478. (Cited on page 118.)

[18] BERTOLI, P., CIMATTI, A., ROVERI, M., AND TRAVERSO, P. Strong planning under partial observability. *Artificial Intelligence 170* (April 2006), 337–384. (Cited on page 118.)

[19] BERTOLI, P., CIMATTI, R., LAGO, U. D., AND PISTORE, M. Extending PDDL to nondeterminism, limited sensing and iterative conditional. In *In Proc. ICAPS03, Workshop on PDDL* (2003), pp. 15–24. (Cited on page 119.)

[20] BERTSEKAS, D. P. *Dynamic Programming and Optimal Control*. Athena Scientific, 2005. (Cited on pages 3 and 36.)

[21] BLAKE, O., BRIDGES, J., CHESTER, E., CLEMMET, J., HALL, S., HANNINGTON, M., HURST, S., JOHNSON, G., LEWIS, S., MALIN, M., MORISON, I., NORTHEY, D., PULLAN, D., RENNIE, G., RICHTER, L., ROTHERY, D., SHAUGHNESSY, B., SIMS, M., SMITH, A., TOWNEND, M., AND WAUGH, L. *Beagle2 Mars: Mission Report.*, 2004. Lander Operations Control Centre, National Space Centre, University of Leicester. (Cited on pages 80, 82 and 83.)

[22] BLONDEL, V. D., AND TSITSIKLIS, J. N. A survey of computational complexity results in systems and control. *Automatica 36*, 9 (2000), 1249–1274. (Cited on page 47.)

[23] BODDY, M. S., AND JOHNSON, D. P. A new method for the global solution of large systems of continuous constraints. In *Global Optimization and Constraint Satisfaction, First International Workshop Global Constraint Optimization and Constraint Satisfaction (COCOS)* (2002), vol. 2861 of *Lecture Notes in Computer Science*, Springer, pp. 142–156. (Cited on page 47.)

[24] BONET, B., AND GEFFNER, H. Planning with incomplete information as heuristic search in belief space. In *Proc. 6th ICAPS* (2000), S. Chien, S. Kambhampati, and C. Knoblock, Eds., AAAI Press, pp. 52–61. (Cited on page 118.)

[25] BONET, B., AND GEFFNER, H. mGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research 24* (2005), 933–944. (Cited on page 118.)

[26] BORRELLI, F. Constrained optimal control for hybrid systems. In *Constrained Optimal Control of Linear and Hybrid Systems*, vol. 290 of *Lecture Notes in Control and Information Sciences*. Springer, 2003, pp. 143–171. (Cited on page 47.)

[27] BOUTILIER, C., DEAN, T., AND HANKS, S. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research 11* (1999), 1–94. (Cited on page 118.)

[28] BRINKSMA, E., AND MADER, A. Verification and optimization of a PLC control schedule. In *Proc. SPIN 2000* (2000), pp. 73–92. (Cited on page 93.)

[29] BRYANT, R. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers C-35*, 8 (Aug 1986), 677–691. (Cited on page 30.)

[30] BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput. 98*, 2 (1992), 142–170. (Cited on pages 1, 2 and 118.)

[31] BUSSIECK, M. R., AND VIGERSKE, S. MINLP solver software, `http://www.matheon.de`, 2010. (Cited on page 36.)

[32] CACHED MURPHI WEB PAGE. `http://www.dsi.uniroma1.it/~tronci/cached.murphi.html`, 2006. (Cited on pages 24 and 52.)

[33] CHESI, G., AND HUNG, Y. Global path-planning for constrained and optimal visual servoing. *IEEE Trans. on Robotics 23*, 5 (2007), 1050–1060. (Cited on page 117.)

[34] CIMATTI, A., CLARKE, E., GIUNCHIGLIA, F., AND ROVERI, M. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer 2* (2000), 2000. (Cited on page 31.)

[35] CIMATTI, A., CLARKE, E. M., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., AND TACCHELLA, A. NuSMV 2: An open-source tool for symbolic model checking. In *CAV* (2002), pp. 359–364. (Cited on page 31.)

[36] CIMATTI, A., PISTORE, M., ROVERI, M., AND TRAVERSO, P. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence 147*, 1-2 (2003), 35 – 84. (Cited on pages 3, 4, 47, 117, 119 and 120.)

[37] Cimatti, A., Roveri, M., and Traverso, P. Strong planning in non-deterministic domains via model checking. In *AIPS* (1998), pp. 36–43. (Cited on pages 46, 121, 123 and 124.)

[38] Clarke, E., and Draghicescu, I. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J. de Bakker, W. de Roever, and G. Rozenberg, Eds., vol. 354 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1989, pp. 428–437. (Cited on page 30.)

[39] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs* (1981), pp. 52–71. (Cited on page 1.)

[40] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. The MIT Press, 1999. (Cited on page 99.)

[41] Coles, A. I., Fox, M., Long, D., and Smith, A. J. Planning with problems requiring temporal coordination. In *Twenty-Second AAAI Conference on Artificial Intelligence* (Vancouver, British Columbia, Canada, July 22-26 2007), AAAI Press., pp. 415–420. (Cited on page 48.)

[42] Coles, A. J., Coles, A. I., Fox, M., and Long, D. Temporal planning in domains with linear processes. In *21st International Joint Conference on Artificial Intelligence (IJCAI)* (Pasadena, California, USA, July 2009), C. Boutilier, Ed., pp. 1671–1676. (Cited on page 48.)

[43] CRISP Research Center web page. `http://www.crisp-org.it`. (Cited on pages 3, 98 and 104.)

[44] CUDD Web Page: `http://vlsi.colorado.edu/~fabio/`, 2009. (Cited on page 55.)

[45] Daniele, M., Traverso, P., and Vardi, M. Y. Strong cyclic planning revisited. In *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning* (London, UK, UK, 2000), ECP ’99, Springer-Verlag, pp. 35–48. (Cited on pages 3, 4 and 119.)

[46] Della Penna, G., Intrigila, B., Lauri, N., and Magazzeni, D. Fast and compact encoding of numerical controllers using obdds. In *Informatics in Control, Automation and Robotics: Selected Papers from ICINCO 2008* (2009), Springer, pp. 75–87. (Cited on page 55.)

[47] Della Penna, G., Intrigila, B., Magazzeni, D., and Mercorio, F. Batch chemical plant PDDL+ model. `http://www.di.univaq.it/gdellape/lamoka/go/?page=chemical`, 2009. (Cited on pages 90 and 92.)

[48] DELLA PENNA, G., INTRIGILA, B., MAGAZZENI, D., AND MERCORIO, F. Repository of planetary lander PDDL+ problems/plans and validation reports., 2009. (Cited on page 86.)

[49] DELLA PENNA, G., INTRIGILA, B., MAGAZZENI, D., AND MERCORIO, F. UP-Murphi: a tool for universal planning on PDDL+ problems. In *Proceedings of The 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)* (2009), AAAI Press, pp. 106–113. (Cited on pages 45, 49, 50, 52, 53 and 63.)

[50] DELLA PENNA, G., INTRIGILA, B., MAGAZZENI, D., AND MERCORIO, F. A PDDL+ benchmark problem: The batch chemical plant. In *Proceedings of The 20th International Conference on Automated Planning and Scheduling (ICAPS 2010)* (Toronto, Canada, 2010), AAAI Press, pp. 222–225. (Cited on page 63.)

[51] DELLA PENNA, G., INTRIGILA, B., MAGAZZENI, D., AND MERCORIO, F. Planning for autonomous planetary vehicles. In *Proceedings of the The Sixth International Conference on Autonomic and Autonomous Systems* (Cancun, Mexico, 2010), pp. 131–136. (Cited on page 63.)

[52] DELLA PENNA, G., INTRIGILA, B., MAGAZZENI, D., AND MERCORIO, F. Resource-optimal planning for an autonomous planetary vehicle. *International Journal of Artificial Intelligence & Applications (IJAIA) 1*, 3 (2010), 15–29. (Cited on page 63.)

[53] DELLA PENNA, G., INTRIGILA, B., MAGAZZENI, D., MERCORIO, F., AND TRONCI, E. Cost-optimal strong planning in non-deterministic domains. in *Proceedings of The 8th International Conference on Informatics in Control, Automation and Robotics (to appear)*. (Cited on pages 123 and 149.)

[54] DELLA PENNA, G., INTRIGILA, B., MELATTI, I., TRONCI, E., AND VENTURINI ZILLI, M. Exploiting transition locality in automatic verification of finite state concurrent systems. *STTT 6*, 4 (2004), 320–341. (Cited on pages 53 and 118.)

[55] DELLA PENNA, G., INTRIGILA, B., MELATTI, I., TRONCI, E., AND ZILLI, M. V. Finite horizon analysis of markov chains with the murphi verifier. *STTT 8*, 4-5 (2006), 397–409. (Cited on page 124.)

[56] DELLA PENNA, G., MAGAZZENI, D., AND MERCORIO, F. A universal planning system for hybrid domains. *Applied Intelligence* (2011), 1–28. 10.1007/s10489-011-0306-z. (Cited on pages 45, 50, 52, 53, 54 and 63.)

[57] DEPARADE, A. A switched continuous model of VHS case study 1. Draft, University of Dortmund, `http://www-verimag.imag.fr/VHS/year1/cs11c.ps`, feb 1999. (Cited on pages 89, 91 and 93.)

[58] DILL, D. L. The mur*phi* verification system. In *CAV* (1996), pp. 390–393. (Cited on page 21.)

[59] DILL, D. L. A retrospective on murphi. *25 Years of Model Checking - LNCS 5000* (2008), 77–88. (Cited on page 21.)

[60] DILL, D. L., DREXLER, A. J., HU, A. J., AND YANG, C. H. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors* (1992), IEEE Computer Society, pp. 522–525. (Cited on page 21.)

[61] DOVIER, A., AND QUINTARELLI, E. Applying Model-checking to solve Queries on semistructured Data. *Computer Languages, Systems & Structures 35*, 2 (2009), 143 – 172. (Cited on page 99.)

[62] EDELKAMP, S. Mixed propositional and numeric planning in the model checking integrated planning system. In *AIPS Workshop on Planning for Temporal Domains* (Toulous, France, April 24 2002), M. Fox and A. M. Coddington, Eds., pp. 47–55. (Cited on page 47.)

[63] EDELKAMP, S. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Intelligence Research 20* (2003), 195–238. (Cited on page 47.)

[64] EDELKAMP, S., AND HELMERT, M. Exhibiting knowledge in planning problems to minimize state encoding length. In *ECP* (1999), pp. 135–147. (Cited on page 119.)

[65] EDELKAMP, S., AND HELMERT, M. MIPS: The model-checking integrated planning system. *AI Magazine 22*, 3 (2001), 67–72. (Cited on page 3.)

[66] EDELKAMP, S., JABBAR, S., AND NAZIH, M. Large-scale optimal PDDL3 planning with MIPS-XXL. In *5th International Planning Competition Booklet. International Conference on Automated Planning and Scheduling* (The English Lake District, Cumbria, UK, 2006), pp. 28–31. (Cited on pages 3 and 47.)

[67] EDELKAMP, S., AND KISSMANN, P. Partial symbolic pattern databases for optimal sequential planning. In *KI* (2008), pp. 193–200. (Cited on page 120.)

[68] EDELKAMP, S., LAFUENTE, A. L., AND LEUE, S. Directed explicit model checking with hsf-spin. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software* (New York, NY, USA, 2001), Springer-Verlag New York, Inc., pp. 57–79. (Cited on page 2.)

[69] EDELKAMP, S., LEUE, S., AND LLUCH-LAFUENTE, A. Directed explicit-state model checking in the validation of communication protocols. *Int. J. Softw. Tools Technol. Transf. 5*, 2 (2004), 247–267. (Cited on page 2.)

[70] EMBURY, S. M., MISSIER, P., SAMPAIO, S., GREENWOOD, R. M., AND PREECE, A. D. Incorporating Domain-Specific Information Quality Constraints into Database Queries. *J. Data and Information Quality 1* (September 2009), 11:1–11:31. (Cited on page 99.)

[71] FAN, W., GEERTS, F., AND JIA, X. A Revival of Integrity Constraints for Data Cleaning. *Proc. VLDB Endow. 1* (August 2008), 1522–1523. (Cited on pages 98 and 99.)

[72] FISHER, C. W., AND KINGMA, B. R. Criticality of data quality as exemplified in two disasters. *Inf. Manage. 39* (December 2001), 109–116. (Cited on page 98.)

[73] FOURMAN, M. Propositional planning. In *Fifth International Conference on Artificial Intelligence Planning and Scheduling - Workshop on Model Theoretic Approaches to Planning (AIPS)* (Breckenridge, CO, USA, 2000), pp. 10–17. (Cited on page 47.)

[74] FOX, M., HOWEY, R., AND LONG, D. Validating plans in the context of processes and exogenous events. In *The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)* (Pittsburgh, Pennsylvania, USA, 2005), AAAI Press / The MIT Press, pp. 1151–1156. (Cited on page 93.)

[75] FOX, M., AND LONG, D. PDDL+: An extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects. *Technical Report, Dept. of Computer Science, University of Durham* (2001). (Cited on pages 47, 80, 82 and 83.)

[76] FOX, M., AND LONG, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. 20* (2003), 61–124. (Cited on pages 3, 14 and 91.)

[77] FOX, M., AND LONG, D. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research 27* (2006), 235–297. (Cited on pages 14, 15 and 54.)

[78] FU, J., NG, V., BASTANI, F. B., AND YEN, I.-L. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. In *IJCAI* (2011), pp. 1949–1954. (Cited on pages 4 and 119.)

[79] FUDENBERG, D., AND TIROLE, J. *Game theory*. MIT Press, aug 1991. (Cited on page 124.)

[80] GEREVINI, A., AND LONG, D. Plan constraints and preferences in PDDL3. *Technical Report, RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia, Italy* (2006). (Cited on page 3.)

[81] GINSBERG, M. L. Ginsberg replies of chapman and schoppers - universal planning research: A good or bad idea? *AI Magazine 10*, 4 (1989), 61–62. (Cited on page 46.)

[82] GIUNCHIGLIA, F., AND TRAVERSO, P. Planning as model checking. In *ECP* (1999), pp. 1–20. (Cited on page 3.)

[83] GIUNCHIGLIA, F., AND TRAVERSO, P. Planning as model checking. In *5th European Conference on Planning: Recent Advances in AI Planning* (London, UK, 2000), Springer-Verlag, pp. 1–20. (Cited on pages 3 and 46.)

[84] GROSSMANN, I. E., AND SAHINIDIS, N. V. Special issue on mixed integer programming and its applications to engineering: Part I. *Optimization and Engineering 3*, 4 (2002). (Cited on pages 3 and 35.)

[85] GROSSMANN, I. E., AND SAHINIDIS, N. V. Special issue on mixed integer programming and its applications to engineering: Part II. *Optimization and Engineering 4*, 1 (2002). (Cited on page 35.)

[86] GRUMBERG, O., AND VEITH, H., Eds. *25 Years of Model Checking - History, Achievements, Perspectives* (2008), vol. 5000 of *Lecture Notes in Computer Science*, Springer. (Cited on page 1.)

[87] HAYATI, S., VOLPE, R., BACKES, P., BALARAM, J., WELCH, R., IVLEV, R., THARP, G., PETERS, S., OHM, T., PETRAS, R., AND LAUBACH, S. The Rocky 7 rover: A Mars sciencecraft prototype. In *in Proceedings IEEE International Conference on Robotics and Automation* (1997), pp. 2458–2464. (Cited on pages 72 and 75.)

[88] HENZINGER, T., HO, P.-H., AND WONG-TOI, H. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer 1*, 1 (dec 1997), 110–122. (Cited on page 32.)

[89] HENZINGER, T. A., HO, P.-H., AND WONG-TOI, H. A user guide to hytech. In *TACAS* (1995), E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, Eds., vol. 1019 of *Lecture Notes in Computer Science*, Springer, pp. 41–71. (Cited on page 32.)

[90] HERRERO, J., BERLANGA, A., MOLINA, J., AND CASAR, J. Methods for operations planning in airport decision support systems. *Applied Intelligence 22* (2005), 183–206. (Cited on pages 3 and 47.)

[91] HOFFMANN, J., AND NEBEL, B. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research 14* (2001), 253–302. (Cited on page 119.)

[92] HOLLDOBLER, S., AND STOR, H. Solving the entailment problem in the fluent calculus using binary decision diagrams. In *Fifth International Conference on Artificial Intelligence Planning Systems (AIPS)* (Breckenridge, CO, USA, 2000), S. Chien, S. Kambhampati, and C. A. Knoblock, Eds., AAAI press, pp. 32–39. (Cited on page 47.)

[93] HOLZMANN, G. J. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991. (Cited on page 1.)

[94] HOLZMANN, G. J. The SPIN model checker. *IEEE Trans. on Software Engineering 23*, 5 (May 1997), 279–295. (Cited on pages 2 and 19.)

[95] HOWEY, R., AND LONG, D. Validating plans with continuous effects. In *In Proc. of the 22nd Workshop of the UK Planning and Scheduling Special Interest Group* (2003). (Cited on pages 63 and 64.)

[96] HOWEY, R., LONG, D., AND FOX, M. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. *ICTAI 00* (2004), 294–301. (Cited on page 63.)

[97] HOWEY, R., LONG, D., AND FOX, M. Validating plans with exogenous events. In *23rd Workshop of the UK Planning and Scheduling Special Interest Group* (University College Cork, Ireland, 2004), pp. 78–87. (Cited on page 93.)

[98] HSU, C. *Cell-to-Cell Mapping*. Springer-Verlag, 1987. (Cited on pages 38 and 40.)

[99] HSU, C. S. A discrete method of optimal control based upon the cell state space concept. *Journal of Optimization Theory and Applications 46* (1985), 547–569. 10.1007/BF00939159. (Cited on page 40.)

[100] HSU, C. S., AND GUTTALU, R. S. An unravelling algorithm for global analysis of dynamical systems - An application of cell-to-cell mappings. *ASME Transactions Series E Journal of Applied Mechanics 47* (Dec. 1980), 940–948. (Cited on pages 38 and 40.)

[101] HSU, C. W., WAH, B. W., HUANG, R., AND CHEN, Y. X. New Features in SGPlan for Handling Soft Constraints and Goal Preferences in PDDL3.0. In *Proc. Fifth International Planning Competition* (June 2006), International Conf. on Automated Planning and Scheduling. (Cited on page 119.)

[102] HU, H., TAI, H., AND SHENOI, S. Incorporating cell map information in fuzzy controller design. In *Proceedings of the 3rd IEEE Conf. on Fuzzy Systems* (1994). (Cited on page 40.)

[103] HUANG, W., WEN, Z., JIANG, Y., AND WU, L. Observation reduction for strong plans. In *Proc. 20th IJCAI* (2007), Morgan Kaufmann, pp. 1930–1935. (Cited on page 118.)

[104] ILCEV, S. D. Antenna systems for mobile satellite applications. In *Microwave Telecommunication Technology, 2009. CriMiCo 2009. 19th International Crimean Conference* (2009), pp. 393 –398. (Cited on page 146.)

[105] IP, C. N., AND DILL, D. L. Better verification through symmetry. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications* (1993), North-Holland, pp. 97–111. (Cited on page 53.)

[106] JENSEN, R. M., AND VELOSO, M. M. Obdd-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research (JAIR) 13* (2000), 189–226. (Cited on pages 4 and 120.)

[107] JENSEN, R. M., VELOSO, M. M., AND BRYANT, R. E. Guided symbolic universal planning. In *ICAPS* (2003), pp. 123–132. (Cited on pages 118 and 120.)

[108] JR., E. M. C., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, 1999. (Cited on page 1.)

[109] KABANZA, F., BARBEAU, M., AND ST-DENIS, R. Planning control rules for reactive agents. *Artificial Intelligence 95* (1997), 67–113. (Cited on page 118.)

[110] KHOUSSAINOV, B., AND NERODE, A. *Automata Theory and Its Applications*. Birkhauser Boston, 2001. (Cited on page 99.)

[111] KISSMANN, P., AND EDELKAMP, S. Solving fully-observable non-deterministic planning problems via translation into a general game. In *KI 2009: Advances in Artificial Intelligence*, B. Mertsching, M. Hund, and Z. Aziz, Eds., vol. 5803 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 1–8. (Cited on pages 4, 118 and 119.)

[112] KISSMANN, P., AND EDELKAMP, S. Gamer, a general game playing agent. *KI 25*, 1 (2011), 49–52. (Cited on pages 4 and 119.)

[113] KOWALEWSKI, S. Description of VHS case study 1: ”Experimental Batch Plant”. `http://astwww.chemietechnik.uni-dortmund.de/~vhs/cs1descr.zip`, July 1998. (Cited on pages 86, 88, 91 and 94.)

[114] KREISSELMEIER, G., AND BIRKHOLZER, T. Numerical nonlinear regulator design. *IEEE Transactions on Automatic Control 39*, 1 (Jan. 1994), 33–46. (Cited on pages 36 and 37.)

[115] KUTER, U., NAU, D. S., REISNER, E., AND GOLDMAN, R. P. Using classical planners to solve nondeterministic planning problems. In *ICAPS* (2008), pp. 190–197. (Cited on pages 4 and 119.)

[116] KWIATKOWSKA, M. Z., NORMAN, G., AND PARKER, D. Probabilistic symbolic model checking with prism: a hybrid approach. *STTT 6*, 2 (2004), 128–142. (Cited on page 124.)

[117] L’AQUILA MODEL CHECKING GROUP. UPMurphi Web Page, `http://www.di.univaq.it/gdellape/lamoka/upmurphi`, 2010. (Cited on page 63.)

[118] LARSEN, K. G., PETTERSSON, P., AND YI, W. UPPAAL in a nutshell. *STTT 1*, 1-2 (1997), 134–152. (Cited on page 2.)

[119] LÉAUTÉ, T., AND WILLIAMS, B. C. Coordinating agile systems through the model-based execution of temporal plans. In *Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)* (Pittsburgh, Pennsylvania, USA, 2005), M. M. Veloso and S. Kambhampati, Eds., AAAI Press / The MIT Press, pp. 114–120. (Cited on pages 3, 47 and 48.)

[120] LEE, D. Design and verification of the Mars exploration rover primary payload. In *Proceedings of Workshop on Spacecraft and Launch Vehicle Dynamic Environments* (2003). (Cited on page 73.)

[121] LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HOROWITZ, M., AND LAM, M. S. The stanford dash multiprocessor. *Computer 25* (March 1992), 63–79. (Cited on page 21.)

[122] LI, H. X., AND WILLIAMS, B. C. Generative planning for hybrid systems based on flow tubes. In *Eighteenth International Conference on Automated Planning and Scheduling (ICAPS)* (Sydney, Australia, 2008), J. Rintanen, B. Nebel, J. C. Beck, and E. A. Hansen, Eds., AAAI Press, pp. 206–213. (Cited on pages 3 and 48.)

[123] MALETIC, J., AND MARCUS, A. Data cleansing: beyond Integrity Analysis. In *Proceedings of the Conference on Information Quality* (2000), pp. 200–209. (Cited on pages 98 and 99.)

[124] MARTINI, M., AND MEZZANZANICA, M. The Federal Observatory of the Labour Market in Lombardy: Models and Methods for the Costruction of a Statistical Information System for Data Analysis. In *Information Systems for Regional Labour Market Monitoring - State of the Art and Prospectives*, C. Larsen, M. Mevius, J. Kipper, and A. Schmid, Eds. Rainer Hampp Verlag, 2009. (Cited on page 104.)

[125] MATTMÜLLER, R., ORTLIEB, M., HELMERT, M., AND BERCHER, P. Pattern database heuristics for fully observable nondeterministic planning. In *ICAPS* (2010), pp. 105–112. (Cited on pages 4, 119 and 120.)

[126] MAUSAM, M., BERTOLI, P., AND WELD, D. S. A hybridized planner for stochastic domains. In *Proc. 20th IJCAI* (2007), Morgan Kaufmann, pp. 1972–1978. (Cited on page 118.)

[127] MAUSAM, M., AND WELD, D. S. Planning with durative actions in stochastic domains. *Journal of Artificial Intelligence Research 31* (January 2008), 33–82. (Cited on page 118.)

[128] MCDERMOTT, D. Reasoning about autonomous processes in an estimated regression planner. In *Thirteenth International Conference on Automated Planning and Scheduling (ICAPS)* (Trento, Italy, 2003), E. Giunchiglia, N. Muscettola, and D. S. Nau, Eds., AAAI Press, pp. 143–152. (Cited on page 47.)

[129] McDermott & the AIPS1998 Planning Competition Committee., D. PDDL: the planning domain definition language. Tech. rep., available at: .: `www.cs.yale.edu/homes/dvm`, 1998. (Cited on pages 3 and 14.)

[130] Meuleau, N., Benazera, E., Brafman, R. I., Hansen, E. A., and Mausam, M. A heuristic search approach to planning with continuous resources in stochastic domains. *Journal of Artificial Intelligence Research (JAIR) 34* (January 2009), 27–59. (Cited on page 118.)

[131] Mezzanzanica, M., Boselli, R., Cesarini, M., and Mercorio, F. Data quality through model checking techniques. In *IDA* (2011), J. Gama, E. Bradley, and J. Hollmén, Eds., vol. 7014 of *Lecture Notes in Computer Science*, Springer, pp. 270–281. (Cited on page 98.)

[132] Mitchell, J. C., Mitchell, M., and Stern, U. Automated analysis of cryptographic protocols using murphi. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1997), IEEE Computer Society, p. 141. (Cited on page 21.)

[133] Mitchell, J. C., Shmatikov, V., and Stern, U. Finite-state analysis of SSL 3.0. In *Seventh USENIX Security Symposium*. USENIX, San Antonio, 1998, pp. 201–216. (Cited on page 21.)

[134] Molineaux, M., Klenk, M., and Aha, D. W. Planning in dynamic environments: Extending HTNs with nonlinear continuous effects. In *Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI)* (Atlanta, Georgia, USA, 2010), M. Fox and D. Poole, Eds., AAAI Press, pp. 1115–1120. (Cited on page 48.)

[135] Müller, H., and Freytag, J.-C. Problems, Methods and Challenges in Comprehensive Data Cleansing. Technical Report HUB-IB-164, Humboldt-Universität zu Berlin, Institut für Informatik, 2003. (Cited on page 98.)

[136] Murphi Tools. `http://www.cs.utah.edu/formal_verification/software/murphi/`. (Cited on page 21.)

[137] Murphi Web Page. `http://sprout.stanford.edu/dill/murphi.html`, 2004. (Cited on pages 21 and 53.)

[138] Nowatzyk, A., Aybay, G., Browne, M. C., Kelly, E. J., Parkin, M., Radke, B., and Vishin, S. The s3.mp scalable shared memory multiprocessor. In *ICPP (1)* (1995), pp. 1–10. (Cited on page 21.)

[139] NuSMV Web Page: `http://nusmv.irst.itc.it/`, 2004. (Cited on pages 2 and 32.)

[140] Papa, M., Tai, H., and Shenoi, S. Cell mapping for controller design and evaluation. *IEEE Control Systems 17*, 2 (1997), 52–65. (Cited on page 40.)

[141] PAPA, M., WOOD, J., AND SHENOI, S. Evaluating controller robustness using cell mapping. *Fuzzy Sets and Systems 121*, 1 (2001), 3–12. (Cited on pages 40 and 146.)

[142] PINTO, J., JOLY, M., AND MORO, L. Planning and scheduling models for refinery operations. *Computers & Chemical Engineering 24*, 9-10 (2000), 2259 – 2276. (Cited on page 36.)

[143] PISTORE, M., AND VARDI, M. Y. The planning spectrum - one, two, three, infinity. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 2003), LICS '03, IEEE Computer Society, pp. 234–. (Cited on pages 3 and 119.)

[144] PROMELA Web Page: `http://spinroot.com/spin/Man/promela.html`. (Cited on page 19.)

[145] REDDY, S. Y., IATAURO, M. J., KÜRKLÜ, E., BOYCE, M. E., FRANK, J. D., AND JÓNSSON, A. K. Planning and monitoring solar array operations on the ISS. In *Eighteenth International Conference on Automated Planning and Scheduling (ICAPS), Scheduling and Planning Applications Workshop (SPARK)* (Sydney, Australia, 2008). (Cited on pages 3 and 47.)

[146] REDMAN, T. C. The impact of poor data quality on the typical enterprise. *Commun. ACM 41* (February 1998), 79–82. (Cited on page 98.)

[147] SAHINIDIS, N., AND GROSSMANN, I. MINLP model for cyclic multiproduct scheduling on continuous parallel lines. *Computers & Chemical Engineering 15*, 2 (1991), 85 – 103. (Cited on page 36.)

[148] SCANNAPIECO, M., MISSIER, P., AND BATINI, C. Data Quality at a Glance. *Datenbank-Spektrum 14* (2005), 6–14. (Cited on page 98.)

[149] SCHMID, U. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, vol. 2654 of *Lecture Notes in Computer Science*. Springer, 2003. (Cited on page 46.)

[150] SCHOPPERS, M. Universal plans of reactive robots in unpredictable environments. In *Proc. IJCAI 1987.* (1987). (Cited on pages 2, 46, 51 and 119.)

[151] SCHUSTER, H. G. *Deterministic Chaos: An Introduction*. Weinheim Physik, 1988. (Cited on page 47.)

[152] SHIN, J.-A., AND DAVIS, E. Processes and continuous change in a SAT-based planner. *Artificial Intelligence 166*, 1-2 (2005), 194–253. (Cited on page 48.)

[153] SMITH, S., AND COMER, D. An algorithm for automated fuzzy logic controller tuning. In *Fuzzy Systems, 1992., IEEE International Conference on* (mar 1992), pp. 615 –622. (Cited on page 146.)

[154] SMV Web Page: `http://www-2.cs.cmu.edu/~modelcheck/smv.html`, 2004. (Cited on page 31.)

[155] SONTAG, E. D. Interconnected automata and linear systems: A theoretical framework in discrete-time. In *Hybrid Systems* (1995), pp. 436–448. (Cited on page 47.)

[156] SPIN Web Page: `http://spinroot.com`, 2004. (Cited on pages 19 and 22.)

[157] STERN, U., AND DILL, D. L. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings* (1995). (Cited on page 21.)

[158] STERN, U., AND DILL, D. L. Improved probabilistic verification by hash compaction. In *CHARME '95: Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (London, UK, 1995), Springer-Verlag, pp. 206–224. (Cited on pages 55 and 60.)

[159] STONE, H. W. Mars Pathfinder microrover: A low-cost, low-power spacecraft. In *Proceedings of the 1996 AIAA Forum on Advanced Developments in Space Robotics* (1996). (Cited on pages 72 and 75.)

[160] STRONG, D. M., LEE, Y. W., AND WANG, R. Y. Data quality in context. *Commun. ACM 40* (May 1997), 103–110. (Cited on page 98.)

[161] T. HENZINGER. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science* (1996), IEEE Computer Society Press, pp. 278–292. (Cited on page 10.)

[162] TATE, E., AND BOYD, S. Finding ultimate limits of performance for hybrid electric vehicles. In *Proceedings of Society of Automotive Engineers 2000 Future Transportation Technology Conference* (2000). (Cited on page 73.)

[163] THE C++ STANDARDS COMMITTEE LIBRARY WORKING GROUP. BOOST library, `http://www.boost.org`, 2011. (Cited on page 107.)

[164] VAN DEN HEEVER, S. A., AND GROSSMANN, I. E. A strategy for the integration of production planning and reactive scheduling in the optimization of a hydrogen supply network. *Computers & Chemical Engineering 27*, 12 (2003), 1813 – 1839. (Cited on page 36.)

[165] VARDI, M. Y. Automata Theory for Database Theoreticians. In *Theoretical Studies in Computer Science*. Academic Press Professional, Inc., 1992, pp. 153–180. (Cited on page 99.)

[166] VERIMAG. ESPRIT-LTR project 26270 (verification of hybrid systems). `http://www-verimag.imag.fr/VHS/`, 2000. (Cited on page 86.)

[167] WASHINGTON, R., GOLDEN, K., BRESINA, J., SMITH, D. E., ANDERSON, C., AND SMITH, T. Autonomous rovers for Mars exploration. In *Proc. IEEE Aerospace Conf.* (1999). (Cited on page 72.)

[168] Wilson, E., Karr, C., and Bennett, J. An adaptive, intelligent control system for slag foaming. *Applied Intelligence 20* (2004), 165–177. (Cited on page 47.)

[169] Yokoi, K., Kanehiro, F., Kaneko, K., Fujiwara, K., Kajita, S., and Hirukawa, H. Experimental study of biped locomotion of humanoid robot hrp-1s. In *Experimental Robotics VIII*, vol. 5 of *Springer Tracts in Advanced Robotics*. Springer, 2003, pp. 75–84. (Cited on page 146.)

[170] Yoon, S. W., Fern, A., and Givan, R. Inductive policy selection for first-order MDPs. In *UAI* (2002), pp. 568–576. (Cited on page 118.)