2016

Motion planning and control: a formal methods approach

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

MOTION PLANNING AND CONTROL: A FORMAL METHODS APPROACH

by

CRISTIAN-IOAN VASILE

B.S., Politehnica University of Bucharest, 2009
M.S., Politehnica University of Bucharest, 2011

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2016

Approved by

First Reader

Calin Belta, PhD
Professor of Mechanical Engineering
Professor of Systems Engineering
Professor of Bioinformatics

Second Reader

John Baillieul, PhD
Distinguished Professor of Mechanical Engineering
Distinguished Professor of Systems Engineering
Distinguished Professor of Electrical and Computer Engineering

Third Reader

Sean Andersson, PhD
Associate Professor of Mechanical Engineering
Associate Professor of Systems Engineering

Fourth Reader

Roberto Tron, PhD
Assistant Professor of Mechanical Engineering
Assistant Professor of Systems Engineering

"Deep in the human unconscious is a pervasive need for a logical universe that makes sense. But the real universe is always one step beyond logic."
– Frank Herbert, Dune

Acknowledgments

First and foremost, I want to thank my advisor, mentor and friend, Professor Calin Belta. His guidance and support helped me grow as a person and researcher. This dissertation is the result of his encouragement, advice, and persistent challenge to surpass myself. I am also grateful for his friendship that made my PhD experience a pleasant one.

I want to thank Mac Schwager for his advice and encouragement thoughtout my time at BU. I would like to thank Igor for his support as a starting PhD student and introducing me to the experimental setup in HyNeSs lab. I want to thank Derya for our long and useful discussions on control and formal methods, and her support and advice. Our fruitful collaboration produced the results presented in Chapters 5 and 7. I want to thank Eric and Kevin for our discussions spanning far too many subject to recount, and fun adventures with the quadrotors and ground robots. Chapter 4 and the experiments in Chapters 6 and 7 are the result of our collaboration. I would also like to thank Vlad for our yearlong lunch conversation, and Prashant and Curtis for our high-spirited and enjoyable meetings while trying to figure out difficult problems in systems and bio-engineering, and game theory. There are many more people that helped and supported me during my PhD, and I am very grateful to have met you: Yushan, Ana Medina, Austin, Joe, Fran, Alphan, Ebru, Guilhem, Maja, Kayhan, Xi, Alyssa, Brian, Trevor, Aamodh, Xiadong, Konstantinos, Iman, Sadra, Demarcus, Rachael, Janos, Dingjiang, Zijian, Junmin.

I would like to extend my gratitude to the members of my dissertation committee, Professors John Baillieul, Sean Andersson, Roberto Tron, and Pirooz Vakili, for their time and valueable comments and suggestions.

Finally, I want to thank my family: my father Liviu, my mother Carmen, my

brother Dan, and my grandmother Doina, and friends: Eduard, Roxana, and Denisa, back in Romania for their unconditional love, support and advice.

MOTION PLANNING AND CONTROL: A FORMAL METHODS APPROACH

CRISTIAN-IOAN VASILE

Boston University, College of Engineering, 2016

Major Professor: Calin Belta, PhD
Professor of Mechanical Engineering
Professor of Systems Engineering
Professor of Bioinformatics

ABSTRACT

Control of complex systems satisfying rich temporal specification has become an increasingly important research area in fields such as robotics, control, automotive, and manufacturing. Popular specification languages include temporal logics, such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL), which extend propositional logic to capture the temporal sequencing of system properties. The focus of this dissertation is on the control of high-dimensional systems and on timed specifications that impose explicit time bounds on the satisfaction of tasks. This work proposes and evaluates methods and algorithms for synthesizing provably correct control policies that deal with the scalability problems. Ideas and tools from formal verification, graph theory, and incremental computing are used to synthesize satisfying control strategies. Finite abstractions of the systems are generated, and then composed with automata encoding the specifications.

The first part of this dissertation introduces a sampling-based motion planning algorithm that combines long-term temporal logic goals with short-term reactive re-

quirements. The specification has two parts: (1) a global specification given as an LTL formula over a set of static service requests that occur at the regions of a known environment, and (2) a local specification that requires servicing a set of dynamic requests that can be sensed locally during the execution. The proposed computational framework consists of two main ingredients: (a) an off-line sampling-based algorithm for the construction of a global transition system that contains a path satisfying the LTL formula, and (b) an on-line sampling-based algorithm to generate paths that service the local requests, while making sure that the satisfaction of the global specification is not affected.

The second part of the dissertation focuses on stochastic systems with temporal and uncertainty constraints. A specification language called Gaussian Distribution Temporal Logic is introduced as an extension of Boolean logic that incorporates temporal evolution and noise mitigation directly into the task specifications. A sampling-based algorithm to synthesize control policies is presented that generates a transition system in the belief space and uses local feedback controllers to break the *curse of history* associated with belief space planning. Switching control policies are then computed using a product Markov Decision Process between the transition system and the Rabin automaton encoding the specification. The approach is evaluated in experiments using a camera network and ground robot.

The third part of this dissertation focuses on control of multi-vehicle systems with timed specifications and charging constraints. A rich expressivity language called Time Window Temporal Logic (TWTL) that describes time bounded specifications is introduced. The temporal relaxation of TWTL formulae with respect to the deadlines of tasks is also discussed. The key ingredient of the solution is an algorithm to translate a TWTL formula to an annotated finite state automaton that encodes all possible temporal relaxations of the given formula. The annotated automata

are composed with transition systems encoding the motion of all vehicles, and with charging models to produce control strategies for all vehicles such that the overall system satisfies the mission specification. The methods are evaluated in simulation and experimental trials with quadrotors and charging stations.

Contents

List of Tables

List of Figures

List of Abbreviations

AST	.....	Abstract Syntax Tree
BLTL	.....	Bounded Linear Temporal Logic
CTL	.....	Computation Tree Logic
DAG	.....	Directed Acyclic Graph
DFA	.....	Deterministic Finite State Automaton
DFS	.....	Depth-First Search
DFW	.....	Disjunction-Free Within Form
DRA	.....	Deterministic Rabin Automaton
DTS	.....	Deterministic Transition System
EC	.....	End Component
EST	.....	Expansive Space Tree
FIRM	.....	Feedback Information Roadmap
GDTL	.....	Gaussian Distribution Temporal Logic
GR(1)	.....	Generalized Reactivity with rank 1
LQG	.....	Linear Quadratic Gaussian
LTI	.....	Linear Time Invariant
LTL	.....	Linear Temporal Logic
LTL_{-X}	.....	LTL without the next operator
MDP	.....	Markov Decision Process
MTL	.....	Metric Temporal Logic
PCTL	.....	Probabilistic CTL
PLTL	.....	Probabilistic LTL
POMDP	.....	Partially Observable Markov Decision Process
PRM	.....	Probabilistic Roadmap
PSTL	.....	Parametric Signal Temporal Logic
PTS	.....	Product Transition System
RRG	.....	Rapidly exploring Random Graph
RRT	.....	Rapidly exploring Random Tree
SCC	.....	Strongly Connected Component
scLTL	.....	Syntactically Co-safe LTL
SRFS	.....	Self-Reachable Final State
STL	.....	Signal Temporal Logic
TR	.....	Temporal Relaxation
TLI	.....	Temporal Logic Inference
TWTL	.....	Time Window Temporal Logic

Chapter 1

Introduction

Motion planning is a fundamental problem in robotics (LaValle, 2006). The goal is to generate a feasible path for a robot to move from an initial to a final configuration while avoiding obstacles. Approaches based on potential fields, navigation functions, and cell decompositions are among the most commonly used (Choset et al., 2005). These, however, become prohibitively expensive in high dimensional configuration spaces. Sampling-based methods were proposed to overcome this limitation. Examples include the probabilistic roadmap (PRM) algorithm proposed by Kavraki et.al. (Kavraki et al., 1996), which is very useful for multi-query problems, but is not well suited for the integration of differential constraints. In (LaValle and Kuffner, 1999), Kuffner and LaValle proposed rapidly-exploring random trees (RRT). These grow randomly, are biased to explore "new" space (LaValle and Kuffner, 1999) (Voronoi bias), and find solutions quite fast. Moreover, PRM and RRT were shown to be probabilistically complete (Kavraki et al., 1996; LaValle and Kuffner, 1999), but not probabilistically optimal (Karaman and Frazzoli, 2011b). Karaman and Frazzoli proposed RRT* and PRM*, the probabilistically optimal counterparts of RRT and PRM in (Karaman and Frazzoli, 2011b).

A recent trend in robot motion planning is the development of computational frameworks that allow for automatic deployment from rich, high-level, temporal logic specifications. As opposed to traditional methods, which only allow to specify a goal position, these frameworks can capture more complex tasks such as sequencing (e.g.,

"Reach A, then B, and then C"), convergence ("Go to A and stay there for all future times"), persistent surveillance ("Visit A, B, and C, in this order, infinitely often"), and more complex logical combinations of the above, such as "Visit A and then B or C infinitely often. Always avoid D. Never go to E unless F was reached before." It was shown that temporal logics, such as Linear Temporal Logic (LTL), Computational Tree Logic (CTL), and μ-calculus, and their probabilistic versions (PLTL, PCTL), can be used as formal languages for motion planning (Kress-Gazit et al., 2007; Wongpiromsarn et al., 2009; Bhatia et al., 2010; Karaman and Frazzoli, 2009; Ding et al., 2011). Adapted model checking algorithms and automata game techniques (Kress-Gazit et al., 2007; Chen et al., 2012) have been used to generate plans and control policies for finite models of robot motion. Such models were obtained through abstractions, which are essentially partitions of the robot configuration space that capture the ability of the robot to steer among the regions in the partition (Belta et al., 2005). As a result, they suffer from the same scalability issues as the cell-based decomposition methods.

Scalability is also an issue when dealing with stochastic systems, where the problem arises due to the *curse of history* that is associated with belief space planning. We use sampling-based techniques to synthesize switched closed-loop control policies for a dynamical system with observation noise while achieving high-level tasks given as temporal logic formulae. Significant observation and actuation noises are inherent in many engineering applications, such as robotics or power networks, in which control actions must be made in real time in response to uncertain or incomplete state and model information. Temporal logic formulae interleave Boolean logic and temporal operators with system properties to specify rich global behaviors. In the domain of robotics, an example of a task that can be encoded in temporal logic is "Periodically clean the living room and then the bathroom. Put the trash in the bin in the kitchen

or outside. Go to a charging station after cleaning is complete. Always avoid the bedroom." In the absence of observation noise, tools from formal synthesis can be used to synthesize control policies that ensure these rich specifications are met (Maly et al., 2013). On the other hand, modern control techniques can be used to synthesize controllers automatically to enforce properties such as "drive the state of the system to a safe set while avoiding unsafe states" under observation and dynamics noise (Kaelbling et al., 1998; van den Berg et al., 2011; Hauser, 2011; Bachrach et al., 2012; Vitus and Tomlin, 2011; Lesser and Oishi, 2015).

Lastly, the routing problem of multi-vehicle systems suffers from scalability issues as well with respect to the number of vehicles. In the Vehicle Routing Problem (VRP), the goal is to find N trajectories for N vehicles achieving a task (e.g., visiting all locations in minimum time). There are various extensions of VRP addressing time capacities, service time windows, service orders (e.g., (Toth and Vigo, 2001; Vasile and Belta, 2014b)), or uncertainty in service requests, travel time, or vehicle availability (e.g., (Bullo et al., 2011; Mu et al., 2011; Chen et al., 2006)). The VRPs are NP-hard combinatorial optimization problems. Typically, finding the optimal trajectories requires to explore all the possible routes. Such an exploration can be achieved by various optimization methods (e.g., integer linear programming, dynamic programming, branch and bound), whose computational complexities increase exponentially with the problem size. This has motivated the development of heuristics or approximate algorithms that result in acceptably good solutions with a lower complexity (e.g., (Laporte, 1992; Pavone et al., 2009)).

In some VRPs, simultaneous visits or relative timings between visiting particular locations might be critical for the task accomplishment. In general, if there exist some tasks that involve a temporal and logical ordering, it is hard to formulate them in the classical optimization setup. Temporal logics (TL) are rich and expressive

specification languages that can be used to address this issue. For example, the authors of (Bhatia et al., 2010) and (Kress-Gazit et al., 2009) address motion planning problems with specifications given in linear temporal logic. Alternatively, a VRP with metric temporal logic formulae is solved in (Karaman and Frazzoli, 2008).

In this dissertation, we tackle planning problems in robotics that pose scalability issues due to: (a) high-dimensional configuration space, (b) stochastic nature, and (c) multi-system structure (i.e., multi-vehicle systems). Another major problem considered is the control of systems from timed specifications. In the third part of the dissertation, we propose a logic called *Time Window Temporal Logic* and an automata-based framework for solving synthesis, verification, and learning problems. We then employ this framework to solve persistent multi-vehicle routing problems in two setting: (1) deterministic models for motion and charging with fixed specification; (2) stochastic charging and relaxed specifications.

1.1 Reactive Temporal Logic Path Planning

In the first part of the dissertation, we address the problem of generating a path for a robot required to satisfy a (global) LTL specification over some known, static service requests, while at the same time servicing a set of locally sensed requests ordered according to their priorities. Consider, for example, a disaster relief scenario requiring an unmanned aircraft to provide persistent surveillance of some affected regions in order to assess the danger posed by unsafe structures with known locations (e.g., by repeatedly taking photos of such regions and uploading the photos at a base region). During flight, by using an onboard camera, the robot looks for survivors and fires. If detected, such requests need to be serviced (e.g., fires need to be extinguished and rescue teams need to be alerted if survivors are detected), possibly with predefined priorities, while making sure that the global, surveillance mission is not compromised.

To address the scalability issues mentioned above, we propose a randomized sampling approach that consists of two components:(1) an off-line algorithm that generates a finite transition system that contains a run satisfying the global specification, and (2) an on-line algorithm that finds local paths that satisfy the local specification, while at the same time making sure that progress is made towards satisfying the global specification.

For the off-line component of the framework, we propose a sampling-based path planning algorithm that finds an infinite path satisfying an LTL formula over a set of properties that hold at some regions in the workspace. The procedure is based on the incremental construction of a transition system in the configuration space followed by the search for one of its satisfying paths. One important feature of the algorithm is that, at a given iteration, it only scales with the number of samples and transitions added to the transitions system at that iteration. This, together with a notion of "sparsity" that we define and enforce on the transition system, play an important role in keeping the overall complexity at a manageable level. In fact, we show that, under some mild assumptions, our definition of sparsity leads to the best possible complexity bound for finding a satisfying path. Finally, while the number of samples increases, the probability that a satisfying path is found approaches 1, i.e., our algorithm is probabilistically complete.

The closest to our proposed off-line algorithm is the work by Karaman and Frazzoli (Karaman and Frazzoli, 2009; Karaman and Frazzoli, 2012), where the specifications are given in deterministic μ-calculus. As in this paper, in (Karaman and Frazzoli, 2009), the authors guarantee probabilistic completeness and scalability with added samples only at each iteration of their algorithm. However, deterministic μ-calculus formulae have unnatural syntax based on fixed point operators, and are difficult to use by untrained human operators. In contrast, Linear Temporal Logic

(LTL) has friendly syntax and semantics, which can be easily translated to natural language (Raman et al., 2013). Note that there is no known procedure to transform an LTL formula ϕ into a μ-calculus formula Ψ such that the size of Ψ is polynomial in the size of ϕ (for details see (Cranen et al., 2011)). In (Karaman and Frazzoli, 2009), the authors employ the fixed point (Knaster-Tarski) theorem to find a satisfying path. Their method is based on maintaining a "product" graph between the transition system and every sub-formula of their deterministic μ-calculus specification and checking for reachability and the existence of a "type" of cycle on the graph. On the other hand, our algorithm maintains the product automaton between the transition system and a Büchi automaton corresponding to the given LTL specification. Note that, as opposed to LTL model checking (Baier and Katoen, 2008), we use a modified version of product automaton that ensures reachability of the final states. Moreover, we impose that the states of the transition system be bounded away from each other (by a given function decaying in terms of the size of the transition system). Sparseness is also explored by Dobson and Berkis in (Dobson and Bekris, 2013) for PRM using different techniques.

The on-line component of our framework uses sampling-based methods as well. However, in this case the focus is on servicing local request and avoiding local obstacles within the bounded sensing area of the robot, while ensuring the satisfaction of the global specification in the long term. The proposed on-line algorithm is based on the definition of a potential function over the global transition system that ensures progress toward satisfaction of the global specification. This idea is inspired from (Ding et al., 2014). The new algorithm that we propose for the computation of the potential function improves the complexity of the algorithm from (Ding et al., 2014) by a polynomial factor. The new algorithm is shown to be correct and to have the same complexity as Dijkstra's algorithm.

The main contribution of this work is a sampling-based, formal framework that combines infinite-time satisfaction of temporal logic global specifications with reactivity to requests sensed locally. Related works include (Maly et al., 2013; Livingston and Murray, 2013; Livingston et al., 2013; Tumova et al., 2013b; Ulusoy et al., 2013a). In (Maly et al., 2013), the authors consider global specifications given in the more restrictive scLTL fragment of LTL. To deal with the state-space explosion problem, they propose a layered path planning approach which uses a cell decomposition of the configuration space for high-level temporal planning and expansive space trees (EST) for kino-dynamic planning of the low-level, cell-to-cell motion. The on-line algorithm from (Tumova et al., 2013b) finds minimum violating paths for a robot when the global specification can not be enforced completely. In (Livingston and Murray, 2013; Livingston et al., 2013), the global specifications are given in the GR(1) fragment of LTL, and on-line local re-planning is done through patching invalidated paths based on μ-calculus specifications. Finally, the idea of using a potential function to enforce the satisfaction of an infinite-time specification through local decisions is inspired from (Ding et al., 2014; Ulusoy et al., 2013a).

1.2 Control in Belief Space with Temporal Logic Specifications

In the second part of the dissertation, we present an automatic, hierarchical control synthesis algorithm that extends tools from formal synthesis and stochastic control to enforce temporal logic specifications. Though our approach is quite general, we use examples from robotic navigation throughout the paper to motivate our approach. We evaluate our algorithm with experiments using a wheeled robot with noisy actuators localized by a noisy, static camera network performing a persistent navigation task.

While synthesizing control policies to enforce temporal logic properties under dy-

namics noise has been extensively considered in the literature (Zamani et al., 2014), observation noise has only recently been considered (Maly et al., 2013; Jones et al., 2013; Leahy et al., 2015; Svorenova et al., 2013; Ayala et al., 2014). One of the technical challenges of incorporating observation noise into formal synthesis is that satisfaction of temporal logic properties is in general defined with respect to the state trajectory of the system rather than the evolution of the belief (as measured by a posterior probability distribution) about this state. In this paper, we introduce the paradigm of Gaussian distribution temporal logic (GDTL) which allows us to specify properties involving the uncertainty in the state of the system, e.g. "Ensure that the uncertainty (measured by variance) of the robot's x position is always below 0.1 m^2". GDTL formulae can be translated to Rabin automata using off-the-shelf tools (Jones et al., 2013).

The problem of synthesizing controllers to enforce a GDTL specification is in general a discrete time, continuous space partially observable Markov decision process (POMDP). Our approach approximates the optimal solution with a computationally feasible hierarchical sampling-based control synthesis algorithm. Most existing sampling-based algorithms sample points directly in belief space (Patil et al., 2015; Burns and Brock, 2007; Bry and Roy, 2011; Prentice and Roy, 2009), which requires synthesizing distribution-to-distribution controllers. Such synthesis problems are computationally difficult and may require significant modeling on the part of a control designer. To circumvent these challenges, we base the core of our algorithm on feedback information roadmaps (FIRMs). The FIRM motion planner extends probabilistic roadmaps (PRMs) (Thrun et al., 2005), to handle observation noise. In FIRM, points are sampled directly in the state space (rather than in belief space) and feedback control policies, e.g. linear quadratic Gaussian (LQG) controllers, stabilize the system about nodes along paths in the roadmap. The behavior of the closed-

loop system is then used to predict how the state estimate evolves. The associated trajectories of the estimate induce a roadmap in the belief space.

If the goal of the problem were only to reach a given region of the belief space, one could construct a switched controller by finding a path in the roadmap from the initial distribution to a node contained within the goal set and then applying the corresponding sequence of controllers. During the application of the controller, however, we do not have any guarantees about whether or not the evolution of the system will violate the given specification. Therefore, we can only estimate with what probability the given controller drives the distribution to the next collection of nodes without violating the specification. This allows us to construct a Markov decision process in which the states correspond to nodes, actions correspond to controller pairs, and transition probabilities correspond to the probability of the closed-loop system reaching the next node without violating the specification. Applying dynamic programming to this system yields a policy that maps the current region of belief states to the pair of controllers to be applied. Combining the policy with the synthesized LQG controllers yields a state-switched feedback controller that satisfies the system specifications with some minimum probability.

Given a Rabin automaton constructed from a GDTL formula and a FIRM, we construct a graph product between the two, called the GDTL-FIRM, to check if the state space has been sampled sufficiently to synthesize a switched controller satisfying the specification with positive probability. We use techniques similar to those in sampling-based formal synthesis work (Agha-mohammadi et al., 2014; Karaman and Frazzoli, 2009; Karaman and Frazzoli, 2012; Vasile and Belta, 2013; Vasile and Belta, 2014a) to construct the GDTL-FIRM incrementally until we find a policy with sufficiently high satisfaction probability.

1.3 Time Window Temporal Logic

Temporal logics provide mathematical formalisms to reason about (concurrent) events in terms of time. Due to their rich expressivity, they have been widely used as specification languages to describe properties related to correctness, termination, mutual exclusion, reachability, or liveness (Manna and Pnueli, 1981). Recently, there has been great interest in using temporal logic formulae in the analysis and control of dynamical systems. For example, linear temporal logic (LTL) (Baier and Katoen, 2008) has been extensively used in motion planning and control of robotic systems, e.g., (Ulusoy et al., 2013c; Karaman and Frazzoli, 2008; Aksaray et al., 2015; Wongpiromsarn et al., 2010; Belta et al., 2005; Wongpiromsarn et al., 2009; Kloetzer and Belta, 2008; Fainekos et al., 2009; Kress-Gazit et al., 2009; Leahy et al., 2014).

In some real-world applications, the tasks may involve some time constraints (e.g., (Solomon, 1987; Pavone et al., 2009)). For example, consider a robot that is required to achieve the following tasks: every visit to A needs to be immediately followed by visiting B within 5 time units; two consecutive visits to A need to be at least 10 time units apart; or visiting A and visiting B need to be completed within 15 time units. Such tasks cannot be described by LTL formulae since LTL cannot deal with temporal properties with explicit time constraints. Therefore, bounded temporal logics are used to capture the time constraints over the tasks. Examples are bounded linear temporal logic (BLTL) (Tkachev and Abate, 2013; Jha et al., 2009), metric temporal logic (MTL) (Koymans, 1990), and signal temporal logic (STL) (Maler and Nickovic, 2004).

We propose a specification language called *time window temporal logic* (TWTL). The semantics of TWTL is rich enough to express a wide variety of time-bounded specifications, e.g., "*monitor* A for 3 time units within the time interval $[0, 5]$ and

after that *monitor B* for 2 time units within $[4, 9]$. This logic was defined in (Vasile and Belta, 2014b; Aksaray et al., 2016), and used to specify persistent surveillance tasks for multi-robot systems. Moreover, we define a notion of *temporal relaxation* of a TWTL formula, which is a quantity computed over the time intervals of a given TWTL formula. In this respect, if the temporal relaxation is: *negative*, then the tasks expressed in the formula should be completed before their designated time deadlines (i.e., satisfying the relaxed formula implies the satisfaction of a more strict formula than the original formula); *zero*, then the relaxed formula is exactly the same as the original formula; *positive*, then some tasks expressed in the formula are allowed to be completed after their original time deadlines (i.e., satisfying the relaxed formula may imply the violation of the original formula or the satisfaction of a less strict formula).

We propose an automata-based framework to solve verification, synthesis, and learning problems that involve TWTL specifications. One property of TWTL specifications we exploit in the proposed solutions is that the associated languages are finite. In the theoretical computer science literature, finite languages and the complexity of constructing their corresponding automata have been extensively studied (Maia et al., 2013; Han and Salomaa, 2007; Câmpeanu et al., 2001; Gao et al., 2011; Daciuk, 2003). One of the main benefits of the proposed framework is its capability to efficiently construct the annotated automata that can encode not only the original formula but also all temporal relaxations of the given formula. Such an efficient construction mainly stems from the proposed algorithms that are specifically developed for TWTL formulae.

The proposed language TWTL has several advantages over existing temporal logics. First, a desired specification can be represented in a more compact and comprehensible way in TWTL than BLTL, MTL, or STL. For example, deadlines expressed in a TWTL formula indicate the exact time bounds as opposed to an STL formula

where the time bounds can be shifted. Consider a specification as "stay at A for 4 time steps within the time window $[0, 10]$", which can be expressed in TWTL as $[H^4 A]^{[0,10]}$. The same specification can be expressed in STL as $F_{[0,10-4]} G_{[0,4]} A$ where the outermost time window needs to be modified with respect to the inner time window. Furthermore, compared to BLTL and MTL, the existence of an explicit concatenation operator results in a more compact representation for serial tasks that are prevalent in various applications including robotics, sensor systems, and manufacturing systems. Under some mild assumptions, we provide a very efficient (linear-time) algorithm to handle concatenation of tasks. In general, the complexity associated with the concatenation operation is exponential in the worst case, even for finite languages (Maia et al., 2013).

Second, the notion of temporal relaxation enables a generic framework to construct the automaton of all possible relaxations of a TWTL formula. In the literature, there are some studies investigating the control synthesis problems for minimal violations of LTL fragments (Reyes Castro et al., 2013; Tumova et al., 2013a; Tumova et al., 2014; Livingston et al., 2013; Guo and Dimarogonas, 2015). In contrast to existing works, the annotated automaton proposed in this paper can encode all possible temporal relaxations of a given formula. Accordingly, such an automaton can be used in a variety of problems related to synthesis, verification, and learning to satisfy minimally relaxed formulae. Third, we show that the complexity of constructing the automata for a given TWTL formula is independent of the corresponding time bounds. To achieve this property, we exploit the structure of finite languages encoded by TWTL formulae.

We present a set of provably-correct algorithms to construct the automaton of a given TWTL formula (both for the relaxed and unrelaxed cases). We formulate a generic problem in terms of temporal relaxation of a TWTL formula, which can be

specialized into problems such as verification, synthesis, and learning. We developed a Python package to solve these three problems, which is available for download from *hyness.bu.edu/twtl*.

1.4 Persistent Vehicle Routing Problem with Charging and Temporal Logic Constraints

The Vehicle Routing Problem (VRP) was first formulated in (Dantzig and Ramser, 1959) as a distribution problem for gasoline from a terminal to service stations using trucks. The basic formulation of VRP is as follows (Karaman and Frazzoli, 2011a; Beck et al., 2003): given N identical vehicles initially located at a depot, a set of sites, and a distance matrix between the sites and the depot, compute a tour for each vehicle such that each tour starts and ends at the depot, every site is visited exactly once, and the overall travelled distance is minimized. The VRP is known to be NP-hard (Garey and Johnson, 1979). Several versions of this problem, which incorporate constraints on the carrying capacity, delivery time frames, and delivery order have been developed (Toth and Vigo, 2001). With particular relevance to this paper is the VRP with Time Windows (VRPTW), in which a service time interval (window) is specified for each site (Toth and Vigo, 2001).

We introduce P-VRP, a persistent surveillance version of VRP. The new problem formulation can be seen as a four-fold extension of a relaxed version of VRPTW, in which no restriction is implicitly assumed about the number of visits to the sites. First, we allow for rich, temporal logic constraints on the order in which sites are to be visited. Second, to accommodate persistent surveillance missions, our problem has infinite-time semantics. For example, in our new, user-friendly specification language, called Time-Window Temporal Logic (TWTL), see Chapter 5, we can describe missions such as "Service sites A, B, and C infinitely often within time windows [2,7],

[6,12], and [5,20], respectively. The service times for A, B, and C are 2, 3, and 1, respectively." [1] Third, we incorporate resource constraints. We assume that, while moving in the environment, each vehicle consumes a resource (e.g., battery charge or fuel) proportionally to the time away from a depot. There is an upper limit on the quantity of the resource each vehicle can store. To replenish their reserves, the vehicles need to return to the depots. Finally, to allow for many revisits to a particular location, we explicitly model and deal with the collision avoidance problem.

Our proposed technical approach brings together concepts and tools from automata theory, formal verification, and optimization. Given a specification as a formula of TWTL, we first translate it to a finite state automaton that accepts the satisfying language. This is then composed with finite transition systems modeling the motion of the vehicles in the environment and the charging constraints. In this product automaton, among all the collision-free motion plans that satisfy the specification and the charging constraints, we select an optimal one. We explore two different optimization criteria. The first is the infinite-time limit of the duration needed for the completion of a surveillance round. The second penalizes the long-term average of the same quantity. These criteria lead to NP-complete problems. We impose some additional restrictions to reduce the problems to manageable sizes. We present simulation case studies and experimental trials with a team of quadrotors involved in a temporal logic persistent surveillance mission with deadlines. The quadrotors can automatically land and charge at a set of fixed charging stations.

This work is related to (and inspired from) several recent works that promote the use of temporal logics and formal methods (Baier and Katoen, 2008) for robot motion planning and control (Kress-Gazit et al., 2007; Wongpiromsarn et al., 2009; Bhatia et al., 2010; Karaman and Frazzoli, 2009; Ding et al., 2011; Smith et al.,

[1] The "classical" VRP constraint that all sites need to be visited exactly once can be easily enforced as a TWTL formula.

2011; Ulusoy et al., 2013c). In particular, (Smith et al., 2011; Ulusoy et al., 2013c) consider optimal persistent surveillance problems with temporal logic constraints and optimality guarantees. However, resource constraints are not considered. In addition, the specification language, which is off-the-shelf LTL, does not capture time windows. Resource constraints for the routing problem restricted to one vehicle and the classical setup of servicing all sites (i.e., no temporal logic specifications) are considered in (Sundar and Rathinam, 2014).

The closest related work is (Karaman and Frazzoli, 2008), which contains a mixed integer linear programming formulation of VRP called VRP-MTL. The specifications are given as formulas in a fragment of Metric Temporal Logic (MTL) (Koymans, 1990), where the temporal operators can only be applied to atomic propositions or their negations. The durations of the transition between sites are fixed and each site can be visited at most once. Our logic, TWTL, strictly contains the MTL fragment used in (Karaman and Frazzoli, 2008). In our approach, a site can be serviced multiple times during a tour if it is required by the specification, and bounds (intervals) on transition durations are allowed. VRL-MTL does not take into account resource constraints related to vehicle movement, such as fuel or battery life, considers a single task over a finite horizon, and optimizes a weighted sum of the distances traveled by the vehicles.

1.5 Dynamic Persistent Vehicle Routing Problem with Charging and Temporal Logic Constraints

This part of the dissertation addresses a persistent VRP involving a group of energy-aware vehicles. The vehicles work together to satisfy a global task infinitely many times. The task is given as a Time-Window Temporal Logic (TWTL) formula over a set of locations. The semantics of TWTL is rich enough to capture a wide variety

of timed temporal logic specifications, e.g., "Service A for 3 time units within [0, 5], and after this, service B for 2 time units within [4, 9]. Within 9 time units, if C is serviced for 2 time units, then D should be serviced for 3 time units." Each vehicle is assumed to have a stochastic fuel consumption model, and it leaves the mission area for refueling when necessary. We propose a decoupled and efficient control policy, in which each vehicle makes an individual decision for refueling whenever it reaches a critical fuel threshold, and a centralized controller plans only the trajectories of the vehicles in the mission area.

This work extends the results from the previous section in three ways. First, the previous method is an off-line strategy, which can not handle uncertainty. Here, due to stochasticity in fuel consumption, we propose an *on-line* control policy that recomputes the trajectories during the mission whenever a change occurs in the number of available vehicles. Second, by decoupling refueling decisions from trajectory planning, the proposed policy exhibits a significantly lower computational complexity than the strategy presented in the previous section. Third, while the previous method returns failure in cases where the given TWTL formula cannot be satisfied, here we allow for satisfaction of minimally relaxed formulae. We quantify the *temporal relaxation* of a TWTL formula and compute trajectories by minimizing it. Accordingly, the resulting trajectories provide the best possible satisfaction performance.

1.6 Contributions

In conclusion, the contributions of the dissertation are the following. In the first part of the dissertation, Chapter 3, a sampling-based, formal framework is proposed that combines infinite-time satisfaction of temporal logic global specifications with reactivity to locally sensed requests. We showed that the planning algorithm is probabilistically complete and the incremental checking procedure for the existence of

satisfying policies has the best possible complexity bound. In the second part of the dissertation, Chapter 4, we propose a sampling-based, formal framework to synthesize feedback policies for stochastic systems with maximum satisfaction probability. Experiments with a ground robot were performed to show the performance of the procedures. In the third part of the dissertation the focus is on temporal specifications with timing constraints, and multi-vehicle routing with limited resources. In Chapter 5, we propose a timed logic called *Time Window Temporal Logic*, temporal relations of TWTL formulae, and an automata-based framework for policy synthesis from relaxed TWTL formulae. Finally, we propose an automata-based framework for *Persistent Vehicle Routing Problems* with temporal and charging constraints. Chapter 6 deals with the deterministic case, where we assume that both the motion and the charging of the vehicles are deterministic, and collision between vehicles is explicitly avoided. We proved that our solution is complete and optimality with respect to two cost functions. In Chapter 7, the setup is changed to account uncertainty in the fuel model. Collisions are penalized in terms of fuel and satisfaction of relaxed TWTL formulae is allowed. The performance of the proposed procedures for both the deterministic and stochastic cases were evaluated in experimental trials.

Chapter 2

Formal Methods Preliminaries

In this chapter, we introduce the notation and briefly review the main concepts from formal languages, automata theory, and formal verification. For a detailed exposition of these topics, the reader is refereed to (Baier and Katoen, 2008; Hopcroft et al., 2006) and the references therein.

Let Σ be a finite set. We denote the cardinality and the power set of Σ by $|\Sigma|$ and 2^{Σ}, respectively. $\emptyset$ denotes the empty set. A *word* over Σ is a finite or infinite sequence of elements from Σ. In this context, Σ is also called an *alphabet*. The length of a word w is denoted by $|w|$ (e.g., $|w| = \infty$ if w is an infinite word). Let k, $i \leq j$ be non-negative integers. The k-th element of w is denoted by w_k, and the sub-word $w_i, \ldots, w_j$ is denoted by $w_{i,j}$. A set of words over an alphabet Σ is called a *language* over Σ. The languages of all finite and infinite words over Σ are denoted by Σ^* and Σ^ω, respectively. These are also called Kleene- and ω-closures, respectively.

Definition 2.1 (Deterministic Transition System, DTS)**.** *A weighted deterministic transition system (DTS) is a tuple $\mathcal{T} = (X, x_0, \Delta, \omega, \Pi, h)$, where:*

- *X is a finite set of states;*
- *$x_0 \in X$ is the initial state;*
- *$\Delta \subseteq X \times X$ is a set of transitions;*
- *$\omega : \Delta \to \mathbb{R}^+$ is a positive weight function;*

- Π *is a set of properties (atomic propositions);*
- $h : X \to 2^{\Pi}$ *is a labeling function.*

We also denote a transition $(x, x') \in \Delta$ by $x \to_{\mathcal{T}} x'$. A *trajectory* (or run) of the system is an infinite sequence of states $\mathbf{x} = x_0 x_1 \ldots$ such that $x_k \to_{\mathcal{T}} x_{k+1}$ for all $k \geq 0$. A state trajectory $\mathbf{x}$ generates an *output trajectory* $\mathbf{o} = o_0 o_1 \ldots$, where $o_k = h(x_k)$ for all $k \geq 0$. The *(generated) language* corresponding to a TS $\mathcal{T}$ is the set of all generated output words, which we denote by $\mathcal{L}(\mathcal{T})$. The absence of inputs (control actions) in a DTS implicitly means that a transition $(x, x') \in \Delta$ can be chosen deterministically at every state x.

A Linear Temporal Logic (LTL) formula over a set of properties Π is defined using standard Boolean operators, $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction), and temporal operators, $\mathbf{X}$ (next), $\mathcal{U}$ (until), $\mathbf{F}$ (eventually), $\mathbf{G}$ (always). The semantics of LTL formulae over Π are given with respect to infinite words over 2^{Π}, such as the output trajectories of the DTS defined above. Any infinite word satisfying a LTL formula can be written in the form of a finite prefix followed by infinitely many repetitions of a suffix. Verifying whether all output trajectories of a DTS with set of propositions Π satisfy an LTL formula over Π is called LTL model checking. LTL formulae can be used to describe rich mission specifications. For example, formula $\mathbf{G}(\mathbf{F}(R_1 \wedge \mathbf{F}R_2) \wedge \neg O_1)$ specifies a persistent surveillance task: "visit regions R_1 and R_2 infinitely many times and always avoid obstacle O_1" (see Figure 2·1). In the first part of the dissertation, we consider a particular fragment of LTL, called LTL_{-X} (Baier and Katoen, 2008), which does not include the $\mathbf{X}$ (next) operator. Formal definitions for the LTL syntax, semantics, and model checking can be found in (Baier and Katoen, 2008).

Definition 2.2 (Büchi Automaton)**.** *A (nondeterministic) Büchi automaton is a*

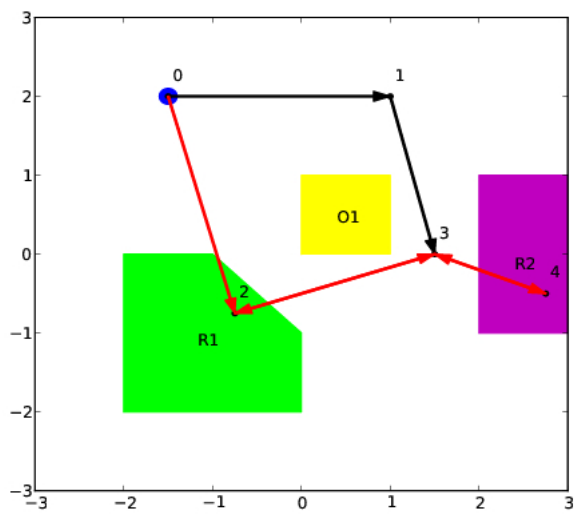


Figure 2·1: A simple map with three features: an obstacle O_1 and two regions of interest R_1 and R_2. The mission specification is $\phi = \mathbf{G}(\mathbf{F}(R_1 \wedge \mathbf{F}R_2) \wedge \neg O_1)$. The initial position of the robot is marked by the blue disk. The graph (in black and red) represents the generated transition system $\mathcal{T}$. The red arrows specify a satisfying trajectory composed of a prefix $[0, 2, 3]$ and infinitely many repetitions of the suffix $[4, 3, 2, 3]$.

tuple $\mathcal{B} = (S_\mathcal{B}, S_{\mathcal{B}_0}, \Sigma, \delta, F_\mathcal{B})$*, where:*

- $S_\mathcal{B}$ *is a finite set of states;*
- $S_{\mathcal{B}_0} \subseteq S_\mathcal{B}$ *is the set of initial states;*
- Σ *is the input alphabet;*
- $\delta : S_\mathcal{B} \times \Sigma \to 2^{S_\mathcal{B}}$ *is the transition function;*
- $F_\mathcal{B} \subseteq S_\mathcal{B}$ *is the set of accepting states.*

A transition $(s, s') \in \delta(s, \sigma)$ is also denoted by $s \xrightarrow{\sigma}_\mathcal{B} s'$. A trajectory of the Büchi automaton $s_0 s_1 \ldots$ is generated by an infinite sequence of symbols $\sigma_0 \sigma_1 \ldots$ if $s_0 \in S_{\mathcal{B}_0}$ and $s_k \xrightarrow{\sigma_k}_\mathcal{B} s_{k+1}$ for all $k \geq 0$. An infinite input sequence over Σ is said to be accepted

by a Büchi automaton $\mathcal{B}$ if it generates at least one trajectory of $\mathcal{B}$ that intersects the set $F_{\mathcal{B}}$ of accepting states infinitely many times.

It is shown in (Baier and Katoen, 2008) that for every LTL formula ϕ over Π there exists a Büchi automaton $\mathcal{B}$ over alphabet $\Sigma = 2^{\Pi}$ such that $\mathcal{B}$ accepts all and only those infinite sequences over Π that satisfy ϕ. There exist efficient algorithms that translate LTL formulae into Büchi automata (Gastin and Oddoux, 2001).

Note, that the converse is not true, there are some Büchi automata for which there is no corresponding LTL formulae. However, there are logics such as deterministic μ-calculus which are in 1-to-1 correspondence with the set of languages accepted by Büchi automata.

Next, deterministic finite state automaton (DFA) are introduced, which are similar to Büchi automata. While Büchi automata may be used to encode general infinite-horizon properties, DFAs may only capture finite-time specifications. However, DFAs have a simpler structure than Büchi automata, and induce simpler verification and synthesis procedures.

Definition 2.3 (Deterministic Finite State Automaton)**.** *A deterministic finite state automaton (DFA) is a tuple* $\mathcal{A} = (S_{\mathcal{A}}, s_0, \Sigma, \delta_{\mathcal{A}}, F_{\mathcal{A}})$*, where:*

- $S_{\mathcal{A}}$ *is a finite set of states;*
- $s_0 \in S_{\mathcal{A}}$ *is the initial state;*
- Σ *is the input alphabet;*
- $\delta_{\mathcal{A}} : S_{\mathcal{A}} \times \Sigma \to S_{\mathcal{A}}$ *is the transition function;*
- $F_{\mathcal{A}} \subseteq S_{\mathcal{A}}$ *is the set of accepting states.*

A transition $s' = \delta_{\mathcal{A}}(s, \sigma)$ is also denoted by $s \xrightarrow{\sigma}_{\mathcal{A}} s'$. A trajectory of the DFA $\mathbf{s} = s_0 s_1 \ldots s_{n+1}$ is generated by a finite sequence of symbols $\boldsymbol{\sigma} = \sigma_0 \sigma_1 \ldots \sigma_n$ if $s_0 \in S_{\mathcal{A}}$

is the initial state of $\mathcal{A}$ and $s_k \xrightarrow{\sigma_k}_{\mathcal{A}} s_{k+1}$ for all $k \geq 0$. The trajectory generated by $\boldsymbol{\sigma}$ is also denoted by $s_0 \xrightarrow{\boldsymbol{\sigma}}_{\mathcal{A}} s_{n+1}$. A finite input word $\boldsymbol{\sigma}$ over Σ is said to be accepted by a finite state automaton $\mathcal{A}$ if the trajectory of $\mathcal{A}$ generated by $\boldsymbol{\sigma}$ ends in a state belonging to the set of accepting states, i.e., $F_{\mathcal{A}}$. A DFA is called *blocking* if the $\delta_{\mathcal{A}}(s, \sigma)$ is a partial function, i.e., the value of the function is not defined for all values in the domain. A blocking automaton rejects words $\boldsymbol{\sigma}$ if there exists $k \geq 0$ such that $s_k \xrightarrow{\sigma_k}_{\mathcal{A}} s_{k+1}$ is not defined. The *(accepted) language* corresponding to a DFA $\mathcal{A}$ is the set of accepted input words, which we denote by $\mathcal{L}(\mathcal{A})$.

Model checking a DTS against an LTL formula is based on the construction of the product automaton between the DTS and the Büchi automaton corresponding to the formula. In this work, we used a modified definition of the product automaton that is optimized for incremental search of a satisfying run. Specifically, the product automaton is defined such that all its states are reachable from the set of initial states.

Definition 2.4 (Product Automaton)**.** *Given a DTS $\mathcal{T} = (X, x_0, \Delta, \omega, \Pi, h)$ and an automaton (Büchi $\mathcal{B}$ or DFA $\mathcal{A}$) $\mathcal{K} = (S_{\mathcal{K}}, S_{\mathcal{K}_0}, 2^{\Pi}, \delta_{\mathcal{K}}, F_{\mathcal{K}})$, their product automaton, denoted by $\mathcal{P} = \mathcal{T} \times \mathcal{K}$, is a tuple $\mathcal{P} = (S_{\mathcal{P}}, S_{\mathcal{P}_0}, \Delta_{\mathcal{P}}, \omega_{\mathcal{P}}, F_{\mathcal{P}})$ where:*

- *$S_{\mathcal{P}_0} = \{x_0\} \times S_{\mathcal{K}_0}$ is the set of initial states (for DFAs there is only one initial state);*

- *$S_{\mathcal{P}} \subseteq X \times S_{\mathcal{K}}$ is a finite set of states which are reachable from some initial state: for every $(x^*, s^*) \in S_{\mathcal{P}}$ there exists a sequence of $\mathbf{x} = x_0 x_1 \ldots x_n x^*$, with $x_k \rightarrow_{\mathcal{T}} x_{k+1}$ for all $0 \leq k < n$ and $x_n \rightarrow_{\mathcal{T}} x^*$, and a sequence $\mathbf{s} = s_0 s_1 \ldots s_n s^*$ such that $s_0 \in S_{\mathcal{K}_0}$, $s_k \overset{h(x_{k+1})}{\rightarrow}_{\mathcal{K}} s_{k+1}$ for all $0 \leq k < n$ and $s_n \overset{h(x^*)}{\rightarrow}_{\mathcal{T}} s^*$;*

- *$\Delta_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$ is the set of transitions, defined by: $((x, s), (x', s')) \in \Delta_{\mathcal{P}}$ iff $x \rightarrow_{\mathcal{T}} x'$ and $s \overset{h(x)}{\rightarrow}_{\mathcal{K}} s'$;*

- $\omega_{\mathcal{P}} : \Delta_{\mathcal{P}} \to \mathbb{R}^+$ *is inherited from* $\mathcal{T}$ *such that* $\omega_{\mathcal{P}}(((x,s),(x',s'))) = \omega((x,x'))$*;*
- $F_{\mathcal{P}} = (X \times F_{\mathcal{K}}) \cap S_{\mathcal{P}}$ *is the set of accepting states of* $\mathcal{P}$*.*

A transition in $\mathcal{P}$ is also denoted by $(x,s) \to_{\mathcal{P}} (x',s')$ if $((x,s),(x',s')) \in \Delta_{\mathcal{P}}$. A trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\ldots$ of $\mathcal{P}$ is a finite or infinite sequence, where $(x_0,s_0) \in S_{\mathcal{P}_0}$ and $(x_k,s_k) \to_{\mathcal{P}} (x_{k+1},s_{k+1})$ for all $k \geq 0$. The acceptance condition is inherited from the specification automaton, Büchi or DFA. A trajectory of $\mathcal{P} = \mathcal{T} \times \mathcal{B}$ is said to be accepting if and only if it intersects the set of final states $F_{\mathcal{P}}$ infinitely many times. A trajectory of $\mathcal{P} = \mathcal{T} \times \mathcal{A}$ is said to be accepting if and only if it ends in a state that belongs to the set of final states $F_{\mathcal{P}}$. It follows by construction that a trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\ldots$ of $\mathcal{P}$ is accepting if and only if the trajectory $s_0 s_1 \ldots$ is accepting in $\mathcal{K}$. As a result, a trajectory of $\mathcal{T}$ obtained from an accepting trajectory of $\mathcal{P}$ satisfies the given specification encoded by $\mathcal{K}$. We denote the projection of a trajectory $\mathbf{p} = (x_0,s_0)(x_1,s_1)\ldots$ onto $\mathcal{T}$ by $\gamma_{\mathcal{T}}(\mathbf{p}) = x_0 x_1 \ldots$. A similar notation is used for projections of finite trajectories. For $x \in X$ and $\mathcal{K} = \mathcal{B}$, we define $\beta_{\mathcal{P}}(x) = \{s \in S_{\mathcal{B}} : (x,s) \in S_{\mathcal{P}}\}$ as the set of Büchi automaton states that correspond to x in $\mathcal{P}$.

For both DTS and automata, we use $|\cdot|$ to denote size, which is the cardinality of the corresponding set of states. A state of a DTS or an automaton is called non-blocking if it has at least one outgoing transition.

Chapter 3

Reactive Temporal Logic Path Planning

In this chapter, we develop a sampling-based motion planning algorithm that combines long-term temporal logic goals with short-term reactive requirements. The mission specification has two parts: (1) a global specification given as a Linear Temporal Logic (LTL) formula over a set of static service requests that occur at the regions of a known environment, and (2) a local specification that requires servicing a set of dynamic requests that can be sensed locally during the execution. The proposed computational framework consists of two main ingredients: (a) an off-line sampling-based algorithm for the construction of a global transition system that contains a path satisfying the LTL formula, and (b) an on-line sampling-based algorithm to generate paths that service the local requests, while making sure that the satisfaction of the global specification is not affected.

The off-line algorithm has four main features. First, it is incremental, in the sense that the procedure for finding a satisfying path at each iteration scales only with the number of new samples generated at that iteration. Second, the underlying graph is sparse, which implies low complexity for the overall method. Third, it is probabilistically complete. Fourth, under some mild assumptions, it has the best possible complexity bound. The on-line algorithm leverages ideas from LTL monitoring and potential functions to ensure progress towards the satisfaction of the global specification while servicing locally sensed requests. Examples and experimental trials illustrating the usefulness and the performance of the framework are included.

3.1 Problem formulation

Consider a robot moving in an environment (workspace) $\mathcal{D}$ containing a set of disjoint regions of interest $\mathcal{R}_G$. We assume that the robot can precisely localize itself in the environment. There is a set of service requests Π_G at the regions in $\mathcal{R}_G$ and their location is given by a map $\mathcal{L}_G : \mathcal{R}_G \to 2^{\Pi_G}$. We assume that these regions as well as the labeling map are static and a priori known to the robot. We will refer to these as *global* regions and requests, because these are used to define the long-term goal of the robot's mission. An example of an environment with global regions and requests is shown in Figure 3·1.

While the robot moves in the environment, it can locally sense a set of dynamic service requests denoted by Π_L and a particular type of avoidance request denoted by π_O, which captures moving obstacles, unsafe areas, etc. We assume $\Pi_G \cap (\Pi_L \cup \{\pi_O\}) = \emptyset$. A dynamic request from Π_L occurs at a point in the environment and has an associated *servicing radius*, which specifies the maximum distance from which the robot can service it. The servicing radius of a request is determined by its type (Π_L) and all servicing radii are known a priori. The robot may service a dynamic request by moving inside the request's servicing radius and performing an appropriate action. We assume that once a request is serviced, it disappears from the environment. The region around the robot in which the robot can sense a dynamic request, including π_O, is called the *sensing area* of the corresponding sensor. For simplicity, we assume that all sensors have the same sensing area. The sensing area may be of any shape and size provided that it is connected and full-dimensional (see Figure 3·1). We assume that the avoidance request π_O is associated with whole regions, parts of which can be detected when they intersect with the robot's sensing area. For simplicity, we refer to regions satisfying π_O as *local obstacles*. The set of regions corresponding to local

obstacles present in the environment at time $t \geq 0$ is denoted by $\mathcal{R}_L(t)$.

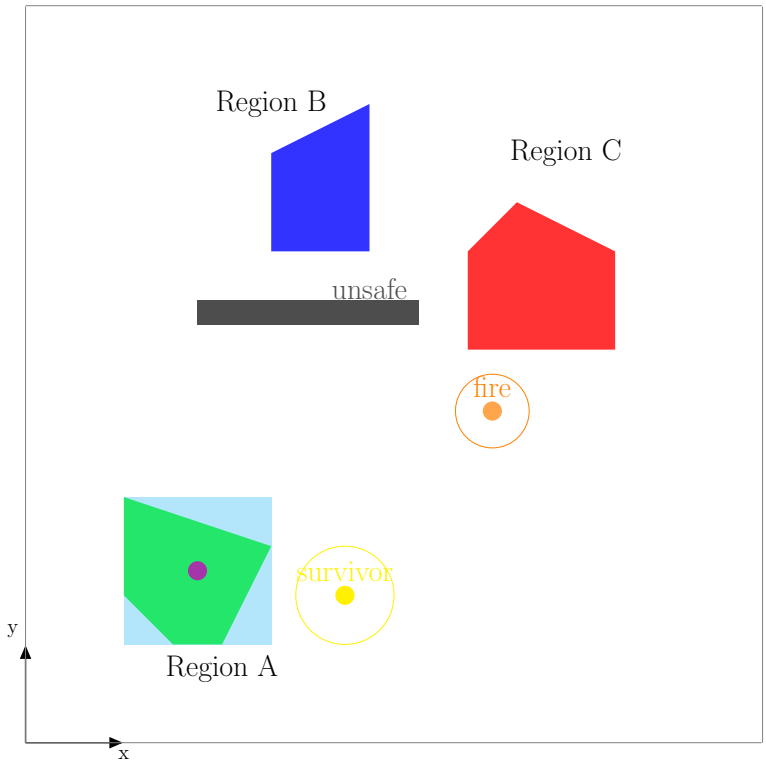


Figure 3·1: Simplified representation of a disaster scenario considered in Example. 3.1. The environment contains three global regions A, B and C colored in green, blue and red, respectively. Three dynamic requests are also shown as colored points: a *survivor* (yellow), a $fire$ (orange), and a local obstacle (black). The circles around them delimit the corresponding servicing areas. The initial position of the robot is shown in magenta and the cyan rectangle corresponds to its sensing area. In this figure the robot does not detect any dynamic request or local obstacles.

The mission specification is composed of two parts: a *global mission specification*, which is defined over the set of global properties Π_G, and a *local mission specification*, which specifies how on-line detected requests Π_L must be handled. The global mission specification, which defines the long-term motion of the robot, is given as an LTL_{-X} formula Φ_G. When the robot passes over a global region, it is assumed that the robot services the requests associated with the region. Therefore, a path traveled

by the robot generates a word over Π_G. A path is said to *satisfy the global mission specification* Φ_G if the corresponding word satisfies Φ_G. The local mission specification is given as a priority function $prio : \Pi_L \to \mathbb{N}$. We assume that $prio$ is an injective function that assigns lower values to higher priority requests. If the robot detects dynamic requests, it must go and service the request with the highest priority. If multiple requests have the same (highest) priority, then the robot can choose any one of them. Also, the robot must avoid all local obstacles marked by π_O.

Planning is performed in the configuration space of the robot. Let $\mathcal{C}$ be the compact configuration space of the robot and $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ be a submersion that maps each configuration x to a position $y = \mathcal{H}(x) \in \mathcal{D}$. Formally, the problem can be formulated as follows:

Problem 3.1 (Reactive Path Planning)**.** *Given a partially known environment described by* $(\mathcal{D}, \mathcal{R}_G, \Pi_G, \mathcal{L}_G, \Pi_L)$*, an initial configuration* $x_0 \in \mathcal{C}$*, an* LTL_{-X} *formula* Φ_G *over the set of properties* Π_G*, and a priority function* $prio : \Pi_L \to \mathbb{N}$*, find an (infinite) path in the configuration space* $\mathcal{C}$ *originating at* x_0 *such that the path* $y = \mathcal{H}(x)$ *in the environment satisfies* Φ_G *and on-line detected dynamic requests, while avoiding local obstacles.*

Example 3.1. *Figure. 3·1 shows a simplified disaster response scenario, in which a fully actuated point robot is deployed in an environment where three global regions of interest A, B and C are defined. The set of dynamic requests is* $\Pi_L = \{fire, survivor\}$ *and the local obstacle is* $\pi_O = unsafe$*. If the robot detects requests* $fire$ *or* $survivor$*, it must service them by going within the corresponding servicing radii and initiating appropriate actions (i.e., extinguishing the fire and providing medical relief, respectively). If the robot detects the local obstacle* $unsafe$ *(shown in black in Figure 3·1), the robot must avoid that region. The limited sensing area of the robot's sensors is depicted in Figure 3·1 by a cyan rectangle.*

The global mission specification is: "Go to region A and then go to regions B or C infinitely often". This specification can be expressed in LTL_{-X} as:

$$\Phi_G := \mathbf{GF}A \wedge \mathbf{G}(A\ \mathcal{U}\ (\neg A\ \mathcal{U}\ (B \vee C))) \tag{3.1}$$

The local mission specification is to "Extinguish fires and provide medical assistance to survivors, with priority given to survivors, while avoiding unsafe areas.". Thus the priority function is defined such that $prio(survivor) = 0$ and $prio(fire) = 1$.

3.1.1 Outline of the Approach

We propose a computational framework to solve Problem 3.1 that consists of two parts: (a) an *off-line* sampling-based algorithm to compute a global transition system $\mathcal{T}_G$ in the configuration space $\mathcal{C}$ of the robot that contains a path whose image in the workspace $\mathcal{D}$ satisfies the global mission specification Φ_G, and (b) an *on-line* sampling-based algorithm that computes at every time step a local control strategy that takes into account dynamic requests such that both local and global mission specifications are met.

A possible approach to the off-line part of Problem 3.1 is to construct a partition of the configuration space such that its image in the workspace contains the regions of interest as elements of the partition. By using input-output linearizations and vector field assignments in the regions of the partition, it was shown that "equivalent" abstractions in the form of finite (not necessarily deterministic) transition systems can be constructed for a large variety of robot dynamics that include car-like vehicles and quadrotors (Belta et al., 2005; Lindemann and LaValle, 2009; Ulusoy et al., 2013b). Model checking and automata game techniques can then be used to control the abstractions from the temporal logic specification (Kloetzer and Belta, 2008). The main limitation of this approach is its high complexity, as both the synthesis

and abstraction algorithms scale at least exponentially with the dimension of the configuration space.

In this paper, we propose a sampling-based algorithm for the construction of $\mathcal{T}_G$ that can be summarized as follows: (1) the LTL formula ϕ_G is translated to the Büchi automaton $\mathcal{B}$; (2) a transition system $\mathcal{T}_G$ is incrementally constructed from the initial configuration x_0 using an RRG-based algorithm; (3) concurrently with (2), the product automaton $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ is updated and used to check if there is a trajectory of $\mathcal{T}_G$ that satisfies Φ_G. As it will become clear later, our proposed algorithm is *probabilistically complete* (LaValle, 2006; Karaman and Frazzoli, 2011b) (i.e., it finds a solution with probability 1 if one exists and the number of samples approaches infinity) and the resulting transition system $\mathcal{T}_G$ is *sparse* (i.e., its states are "far" away from each other). Also, it is incremental, in the sense that its complexity scales only with the number of samples generated at the current iteration, rather than with size of $\mathcal{T}_G$.

The proposed approach to the on-line part of Problem 3.1 is based on the RRT algorithm, a probabilistically complete sampling-based path planning method. RRT randomly grows trees instead of general graphs. We modify the standard RRT in order to find local paths which preserve the satisfaction of the global specification Φ_G, while servicing on-line requests and avoiding locally sensed obstacles. We use ideas from (Bauer et al., 2011) on monitors for LTL formulae and (Ding et al., 2014) on potential functions to ensure the correctness of the local random paths with respect to Φ_G.

3.2 Solution

In the following, we will denote by $\mathcal{T}_G = (X_G, x_0, \Delta_G, \omega_G, \Pi_G, h_G)$ the global transition system, by $\mathcal{B}$ the Büchi automaton encoding the LTL_{-X} formula Φ_G and by $\mathcal{P}_G = \mathcal{T}_G \times$

$\mathcal{B}$ their product. The local transition system is given by $\mathcal{T}_L = (X_L, x_c, \Delta_L, \omega_L, \Pi_L \cup \{\pi_O\}, h_L)$ which is incrementally generated at each time step of the on-line procedure (see Section 3.2.2) from the current configuration x_c. An element of $\mathcal{D}$ will be called a *position*. The states of $\mathcal{T}_G$ and $\mathcal{T}_L$ are configurations in $\mathcal{C}$. The weight of a transition of $\mathcal{T}_G$ or $\mathcal{T}_L$ is given by the distance between its endpoints in $\mathcal{C}$. The labeling function $h_G(x)$, $x \in X_G$, is defined as the proposition set corresponding to the region the projection of x belong to. Formally, $h_G(x) = \mathcal{L}_G(R)$ if $\mathcal{H}(x) \in R$ for some $R \in \mathcal{R}_G$, and $h_G(x) = \emptyset$ otherwise. Similarly, the labeling function $h_L(x)$, $x \in X_L$ is defined as the set of local requests which are satisfied at position $y = \mathcal{H}(x)$ if $y \notin \mathcal{R}_L(t)$, and $h_L(x) = \pi_O$, otherwise. Recall that the robot has knowledge only about the local requests and obstacles inside its sensing area, which is determined by the current position $\mathcal{H}(x_c)$. Also, $h_L(x)$ may be $\emptyset$ if no local requests are satisfied by the corresponding position $y = \mathcal{H}(x)$ and y does not fall inside a local obstacle.

We make the following additional assumptions that are necessary in the technical treatment below. For a set $R \subseteq \mathcal{D}$ that is connected and has full dimension in $\mathcal{D}$, we assume that the inverse set $\mathcal{H}^{-1}(R)$ also has full dimension in $\mathcal{C}$. The global regions and local obstacles are connected sets with non-empty interior, (i.e. they have full dimension in $\mathcal{D}$). Also, all the connected regions in the free space, between global regions and obstacles, respectively, are full dimensional. This implies that all global regions, local obstacles, service areas for dynamic requests, and connected free space regions (all subsets of $\mathcal{D}$) have corresponding inverse sets (through $\mathcal{H}^{-1}$) of non-zero Lebesgue measure in $\mathcal{C}$. It is important to note that these are just technical assumptions, which are normally made in sampling-based approaches, and we do not need to construct the inverse map $\mathcal{H}^{-1}$. In the sampling-based algorithms described below, we only need to check how the environment image of a configuration satisfies features of interest in the environment. Finally, we assume that the robot knows its

configuration precisely and it can follow trajectories in the configuration space made of connected line segments. A path $\mathbf{x}$ in $\mathcal{C}$ is said to satisfy the specification Φ_G if the corresponding path $\mathbf{y} = \mathcal{H}(\mathbf{x})$ in $\mathcal{D}$ satisfies Φ_G. The initial configuration x_0 of the robot is known and $\mathcal{H}(x_0) = y_0$.

3.2.1 Off-line Algorithm

The starting point for our solution to Problem 3.1 is the off-line algorithm to generate the global transition system $\mathcal{T}_G$. The algorithm is based on the RRG algorithm, which is an extension of RRT (Karaman and Frazzoli, 2011b) that maintains a digraph instead of a tree, and can therefore be used as a model for general ω-regular languages (Karaman and Frazzoli, 2009). However, we modify the RRG to obtain a "sparse" transition system that satisfies a given LTL formula Φ_G. More precisely, a transition system $\mathcal{T}_G$ is "sparse" if the minimum distance between any two states of $\mathcal{T}$ is greater than a prescribed function dependent only on the size of $\mathcal{T}_G$ ($\min_{x,x' \in \mathcal{T}_G} \|x - x'\|_2 \geq \eta(|\mathcal{T}_G|)$). The distance used to define sparsity is inherited from the underlying configuration space and is not related to the graph theoretical distance between states in $\mathcal{T}_G$. Throughout this chapter, we will assume that this distance is Euclidean.

As stated in Section 3.1.1, sparsity of $\mathcal{T}_G$ is desired because the transition system is then used in the on-line part of the framework. The environment is partially known by the robot before the start of the mission. Since transitions of $\mathcal{T}_G$ may need to be locally re-planned on-line, $\mathcal{T}_G$ must only capture the essential features of $\mathcal{D}$ such that Φ_G is satisfied. Sparseness also plays an important role in establishing the complexity bounds for the incremental search algorithm (see Section Incremental search for a satisfying run).

Primitive functions

We first briefly introduce the functions used by the algorithm.

Sampling function The algorithm has access to a sampling function *sample* : $\mathbb{N} \rightarrow \mathcal{C}$, which generates independent and identically distributed samples from a given distribution P. We assume that the support of P is the entire configuration space $\mathcal{C}$.

Steer function The steer function *steer* : $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is defined based on the robot's dynamics. [1] Given a configuration x and goal configuration x_g, it returns a new configuration x_n that can be reached from x by following the dynamics of the robot and that satisfies $\|x_n - x_g\|_2 < \|x - x_g\|_2$. If a third parameter η is given, then x_n must be within η distance away from x, $\|x_n - x\|_2 < \eta$.

Near function $near : \mathcal{C} \times \mathbb{R} \rightarrow 2^X$ is a function of a configuration x and a parameter η, which returns the set of states from the transition system $\mathcal{T}_G$ that are at most at η distance away from x. In other words, *near* returns all states in $\mathcal{T}_G$ that are inside the n-dimensional sphere of center x and radius η.

Far function $far : \mathcal{C} \times \mathbb{R} \times \mathbb{R} \rightarrow 2^X$ is a function of a configuration x and two parameters η_1 and η_2. It returns the set of states from the transition system $\mathcal{T}_G$ that are at most at η_2 distance away from x. However, the difference from the *near* function is that far returns an empty set if any state of $\mathcal{T}_G$ is closer to x than η_1. Geometrically, this means that far returns a non-empty set for a given state x if there are states in $\mathcal{T}_G$ which are inside the n-dimensional sphere of center x and radius η_2

[1] In this paper, we will assume that we have access to such a function. For more details about planning under differential constraints see (LaValle, 2006).

and all states of $\mathcal{T}_G$ are outside the sphere with the same center, but radius η_1. Thus, x has to be "far" away from all states in its immediate neighborhood (see Figure 3·2). This function is used to achieve the "sparseness" of the resulting transition system.

isSimpleSegment function $isSimpleSegment : \mathcal{C} \times \mathcal{C} \rightarrow \{0, 1\}$ is a function that takes two configurations x_1, x_2 in $\mathcal{C}$ and returns 1 if the line segment $[x_1, x_2]$ ($\{x \in \mathbb{R}^n : x = \lambda x_1 + (1 - \lambda)x_2, \lambda \in [0, 1]\}$) is simple, otherwise it returns 0. Let $y_1 = \mathcal{H}(x_1)$, $y_2 = \mathcal{H}(x_2)$ and $[y_1, y_2] = \mathcal{H}([x_1, x_2])$ be the projections of x_1, x_2 and the line segment $[x_1, x_2]$ onto the workspace $\mathcal{D}$, respectively. A line segment $[x_1, x_2]$ is simple if $[x_1, x_2] \subset \mathcal{C}$ and the number of times $[y_1, y_2]$ crosses the boundary of any region $R \in \mathcal{R}$ is at most one. Therefore, $isSimpleSegment$ returns 1 if either: (1) y_1 and y_2 belong to the same region R and $[y_1, y_2]$ does not cross the boundary of R or (2) y_1 and y_2 belong to two regions R_1 and R_2, respectively, and $[y_1, y_2]$ crosses the common boundary of R_1 and R_2 once. R or at most one of R_1 and R_2 may be a free space region (a connected set in $\mathcal{D} \setminus \bigcup_{R \in \mathcal{R}} R$). See Figure 3·2 for examples. In Algorithm 1, a transition is rejected if it corresponds to a non-simple line segment (i.e. $isSimpleSegment$ function returns 0).

Bound functions $\eta_1 : \mathbb{Z}_+ \rightarrow \mathbb{R}$ (lower bound) and $\eta_2 : \mathbb{Z}_+ \rightarrow \mathbb{R}$ (upper bound) are functions that define the bounds on the distance between a configuration in $\mathcal{C}$ and the states of the transition system $\mathcal{T}_G$ in terms of the size of $\mathcal{T}_G$. These are used as parameters for functions far and $near$. We impose $\eta_1(k) < \eta_2(k)$ for all $k \geq 1$. We also assume that $c\,\eta_1(k) > \eta_2(k)$, for some finite $c > 1$ and all $k \geq 0$. Also, η_1 tends to 0 as k tends to infinity. The rate of decay of $\eta_1(\cdot)$ has to be fast enough such that a new sample may be generated. Specifically, the set of all configurations where the center of an n-sphere of radius $\eta_1/2$ may be placed such that it does not intersect any of the d-spheres corresponding to the states in $\mathcal{T}_G$ has to have non-zero

measure with respect to the probability measure P used by the sampling function. One conservative upper bound is $\eta_1(k) < \frac{1}{\sqrt{\pi}} \sqrt[d]{\frac{\mu(\mathcal{C})\Gamma(d/2+1)}{k}}$ for all $k \geq 1$, where $\mu(\mathcal{C})$ is the total measure (volume) of the configuration space, d is the dimension of $\mathcal{C}$, and Γ is the gamma function. This bound corresponds to the case when $\mathcal{C}$ is convex and there is enough space to insert an n-sphere of radius $\eta_1/2$ between every two distinct states of $\mathcal{T}_G$. To simplify the notation, we drop the parameter for these functions and assume that k is always given by the current size of the transition system, $k = |\mathcal{T}|$.

Sparse RRG

The goal of the modified RRG algorithm (see Algorithm 1) is to find a satisfying run, but such that the resulting transition system is "sparse", i.e. states are "sufficiently" apart from each other. The algorithm iterates until a satisfying run originating in x_0 is found.

At each iteration, a new sample x_r is generated (line 6 in Algorithm 1). For each state x in $\mathcal{T}_G$ which is "far" from the sample x_r ($x \in far(x_r, \eta_1, \eta_2)$), a new configuration x'_r is computed such that the robot can be steered from x to x'_r and the distance to x_r is decreased (line 10). The two loops of the algorithm (lines 7–13 and 16–21) are executed if and only if the far function returns a non-empty set. However, x'_r is regarded as a potential new state of $\mathcal{T}_G$, and not x_r. Thus, the *steer* function plays an important role in the "sparsity" of the final transition system. Next, it is checked if the potential new transition (x, x'_r) is a simple segment (line 9). It is also verified if x'_r may lead to a solution, which is equivalent to testing if x'_r induces at least one non-blocking state in $\mathcal{P}_G$ (see Algorithm 2). If configuration x'_r and the corresponding transition (x, x'_r) pass all tests, then they are added to the list of new states and list of new transitions of $\mathcal{T}_G$, respectively (lines 12–13).

After all "far" neighbors of x_r are processed, the transition system is updated.

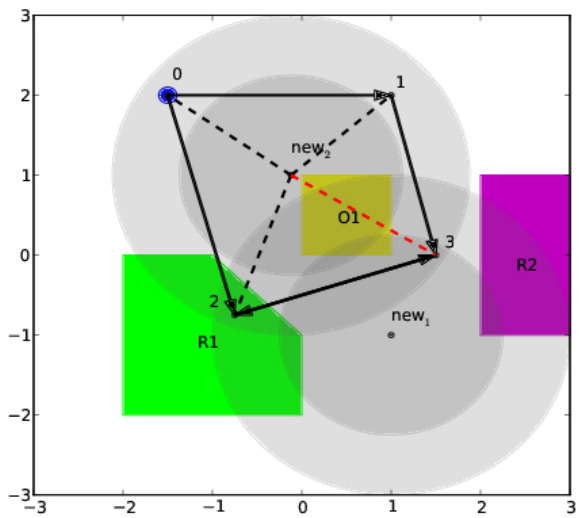


Figure 3·2: A simple map with three features: an obstacle O_1 and two regions R_1, R_2. The robot is assumed to be a fully actuated point and $\mathcal{C} = \mathcal{D} \subset \mathbb{R}^2$. At the current iteration the states of $\mathcal{T}_G$ are $\{0, 1, 2, 3\}$. The transitions of $\mathcal{T}_G$ are represented by the black arrows. The initial configuration is 0 and is marked by the blue disk. The radii of the dark gray (inner) disks and the light gray (outer) disks are η_1 and η_2, respectively. A new sample $new_1 \in \mathcal{C}$ is generated, but it will not be considered as a potential new state of $\mathcal{T}_G$, because it is within η_1 distance from state 3 ($far(new_1, \eta_1, \eta_2) = \emptyset$). Another sample $new_2 \in \mathcal{C}$ is generated, which is at least η_1 distance away from all states in $\mathcal{T}_G$. In this case, $far(new_2, \eta_1, \eta_2) = \{0, 1, 2, 3\}$ and the algorithm attempts to create transition to and from the new sample new_2. The transitions $\{(new_2, 0), (0, new_2), (new_2, 1), (1, new_2), (new_2, 2), (2, new_2)\}$ (marked by black dashed lines) are added to $\mathcal{T}_G$, because all these transitions correspond to simple line segments ($isSimpleSegment$ returns 1 for all of them). For example, $isSimpleSegment(new_2, 0) = 1$, because new_2 and 0 belong to the same region (the free space region) and $[new_2, 0]$ does not intersect any other region. $isSimpleSegment(new_2, 2) = 1$, because $[new_2, 2]$ crosses the boundary between the free space region and region R_1 once. On the other hand, the transitions $\{(new_2, 3), (3, new_2)\}$ (marked by red dashed lines) are not added to $\mathcal{T}_G$, since they pass over the obstacle O_1. In this case, $isSimpleSegment(3, new_2) = 0$, because 3 and new_2 are in the same region, but $[3, new_2]$ crosses the boundary of O_1 twice.

Note that at this point $\mathcal{T}_G$ was only extended with states that explore "new space". However, in order to model ω-regular languages the algorithm must also close cycles. Therefore, the same procedure as before (lines 7–14) is also applied to the newly added states X'_G (lines 15–21 of Algorithm 1). The difference is that it is checked if states from X'_G can steer the robot back to states in $\mathcal{T}_G$ in order to close cycles. Also, because we know that the states in X'_G are "far" from their neighbors, the *near* function will be used instead of the far function. The algorithm returns a (prefix, suffix) pair in $\mathcal{T}_G$ obtained by projection from the corresponding path ($p_0 \xrightarrow{*}_{\mathcal{P}_G} p_F$) and cycle ($p_F \xrightarrow{+}_{\mathcal{P}_G} p_F$) in $\mathcal{P}_G$, respectively. The $*$ above the transition symbol means that the length of the path can be 0 or more, while $+$ denotes that the length of the cycle must be at least 1.

In the end, the result is a transition system $\mathcal{T}_G$ which captures the general topology of the environment. In the next section, we will show that $\mathcal{T}_G$ also yields a run that satisfies the given specification.

Incremental search for a satisfying run

The proposed approach of incrementally constructing a transition system raises the problem of how to efficiently check for a satisfying run at each iteration. As mentioned in the previous section, the search for satisfying runs is performed on the product automaton. Note that testing whether there exists a trajectory of $\mathcal{T}_G$ from the initial configuration x_0 that satisfies the given LTL_{-X} formula Φ_G is equivalent to searching for a path from an initial state p_0 to a final state p_F in the product automaton $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ and for a cycle containing p_F of length greater than 1, where $\mathcal{B}$ is the Büchi automaton corresponding to Φ_G. If such a path and a cycle are found then their projection onto $\mathcal{T}_G$ represents a satisfying infinite trajectory (line 23 of Algorithm 1). Testing whether p_F belongs to a non-degenerate cycle (length greater than 1) is

Algorithm 1: Sparse RRG

Input: $\mathcal{B}$ – Büchi automaton corresponding to Φ_G
Input: x_0 initial configuration of the robot
Output: (prefix, suffix) in $\mathcal{T}_G$

```
Construct T_G with x_0 as initial state
Construct P_G = T_G × B
Initialize scc(·)
while ¬(x_0 ⊨ φ) (≡ ¬(∃p ∈ F_{P_G} s.t. |scc(p)| > 1)) do
    X'_G ← ∅, Δ'_G ← ∅
    x_r ← sample()
    foreach x ∈ far(x_r, η_1, η_2) do
        x'_r ← steer(x, x_r)
        if isSimpleSegment(x_r, x'_r) then
            added ← updatePA(P_G, B, (x, x'_r))
            if added is True then
                X'_G ← X'_G ∪ {x'_r}
                Δ'_G ← Δ'_G ∪ {(x, x'_r)}
    T_G ← T_G ∪ (X'_G, Δ'_G)
    Δ'_G ← ∅
    foreach x'_r ∈ X'_G do
        foreach x ∈ near(x'_r, η_2) do
            if (x = steer(x'_r, x)) ∧ isSimpleSegment(x'_r, x) then
                added ← updatePA(P_G, B, (x, x'_r))
                if added is True then
                    Δ'_G ← Δ'_G ∪ {(x'_r, x)}
    T_G ← T_G ∪ (X'_G, Δ'_G)
return (γ_{T_G}(p_0 →*_{P_G} p_F), γ_{T_G}(p_F →+_{P_G} p_F)), where p_F ∈ F_P
```

equivalent to testing if p_F belong to a non-trivial strongly connected component – SCC (the size of the SCC is greater than 1). Checking for a satisfying trajectory in $\mathcal{P}_G$ is performed incrementally as the transition system is modified.

The reachability of the final states from initial ones in $\mathcal{P}_G$ is guaranteed by construction (see Definition 2.4). However, we need to define a procedure (see Algorithm 2) to incrementally update $\mathcal{P}_G$ when a new transition is added to $\mathcal{T}_G$. Consider

the (non-incremental) case of constructing $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$. This is done by a traversal of $\bar{\mathcal{P}}_G = (X_G \times S_{\mathcal{B}}, \bar{\Delta}_{\mathcal{P}_G})$ from all initial states, where $((x,s),(x',s')) \in \bar{\Delta}_{\mathcal{P}_G}$ if $x \rightarrow_{\mathcal{T}_G} x'$ and $s \overset{h_G(x)}{\rightarrow} s'$. $\bar{\mathcal{P}}_G$ is a product automaton but without the reachability requirement. This suggests that the way to update $\mathcal{P}_G$ when a transition (x,x') is added to $\mathcal{T}_G$, is to do a traversal from all states p of $\mathcal{P}_G$ such that $\gamma_{\mathcal{T}_G}(p) = x$. Also, it is checked if x' induces any non-blocking states in $\mathcal{P}_G$ (lines 1-3 of Algorithm 2). The test is performed by computing the set $S'_{\mathcal{P}_G}$ of non-blocking states of $\mathcal{P}_G$ (line 1) such that $p' \in S'_{\mathcal{P}_G}$ has $\gamma_{\mathcal{T}_G}(p') = x'$ and p' is obtained by a transition from $\{(x,s) : s \in \beta_{\mathcal{P}_G}(x)\}$. If $S'_{\mathcal{P}_G}$ is empty then the transition (x,x') of $\mathcal{T}_G$ is discarded and the procedure stops (line 3). Otherwise, the product automaton $\mathcal{P}_G$ is updated recursively to add all states that become reachable because of the states in $S'_{\mathcal{P}_G}$. The recursive procedure is performed from each state in $S'_{\mathcal{P}_G}$ as follows: if a state p (line 9) is not in $\mathcal{P}_G$, then it is added to $\mathcal{P}_G$ together with all its outgoing transitions (line 10) and the recursive procedure continues from the outgoing states of p; if p is in $\mathcal{P}_G$ then the traversal stops, but its outgoing transitions are still added to $\mathcal{P}_G$ (line 14). The incremental construction of $\mathcal{P}_G$ has the same overall complexity as constructing $\mathcal{P}_G$ from the final $\mathcal{T}_G$ and $\mathcal{B}$, because the recursive procedure just performs traversals that do not visit states already in $\mathcal{P}_G$. Thus, we focus our complexity analysis on the next step of the incremental search algorithm.

The second part of the incremental search procedure is concerned with maintaining the strongly connected components (SCCs) of $\mathcal{P}_G$ (line 14 of Algorithm 2) as new transitions are added (these are stored in $\Delta'_{\mathcal{P}_G}$ in Algorithm 2). To incrementally maintain the SCCs of the product automaton, we employ the soft-threshold-search algorithm presented in (Haeupler et al., 2012). The algorithm maintains a topological order of the super-vertices corresponding to each SCC. When a new transition is added to $\mathcal{P}_G$, the algorithm proceeds to re-establish a topological order and merges vertices if

Algorithm 2: Incremental Search for a Satisfying Run

Input: $\mathcal{P}_G$ – product automaton
Input: $\mathcal{B}$ – Büchi automaton
Input: (x, x') – new transition in $\mathcal{T}_G$
Output: Boolean value – indicates if $\mathcal{P}_G$ was modified

1 $S'_{\mathcal{P}_G} \leftarrow \{(x', s') : s \overset{h_G(x)}{\rightarrow}_{\mathcal{B}} s', s \in \beta_{\mathcal{P}_G}(x), s' \text{ non-blocking}\}$
2 $\Delta'_{\mathcal{P}_G} \leftarrow \{((x, s), (x', s')) : s \in \beta_{\mathcal{P}_G}(x), s \overset{h_G(x)}{\rightarrow}_{\mathcal{B}} s',\ (x', s') \in S'_{\mathcal{P}_G}\}$
3 **if** $S'_{\mathcal{P}_G} \neq \emptyset$ **then**
4 $\quad \mathcal{P}_G \leftarrow \mathcal{P}_G \cup (S'_{\mathcal{P}_G}, \Delta'_{\mathcal{P}_G})$
5 $\quad stack \leftarrow S'_{\mathcal{P}_G}$
6 $\quad$ **while** $stack \neq \emptyset$ **do**
7 $\quad\quad p_1 = (x_1, s_1) \leftarrow stack.pop()$
8 $\quad\quad$ **foreach** $p_2 \in \{(x_2, s_2) : x_1 \rightarrow_{\mathcal{T}_G} x_2, s_1 \overset{h_G(x_1)}{\rightarrow}_{\mathcal{B}} s_2\}$ **do**
9 $\quad\quad\quad$ **if** $p_2 \notin S_{\mathcal{P}_G}$ **then**
10 $\quad\quad\quad\quad \mathcal{P}_G \leftarrow \mathcal{P}_G \cup (\{p_2\}, \{(p_1, p_2)\})$
11 $\quad\quad\quad\quad \Delta'_{\mathcal{P}_G} \leftarrow \Delta'_{\mathcal{P}_G} \cup \{(p_1, p_2)\}$
12 $\quad\quad\quad\quad stack \leftarrow stack \cup \{p_2\}$
13 $\quad\quad\quad$ **else if** $(p_1, p_2) \notin \Delta_{\mathcal{P}_G}$ **then**
14 $\quad\quad\quad\quad \Delta_{\mathcal{P}_G} \leftarrow \Delta_{\mathcal{P}_G} \cup \{(p_1, p_2)\}$
15 $\quad\quad\quad\quad \Delta'_{\mathcal{P}_G} \leftarrow \Delta'_{\mathcal{P}_G} \cup \{(p_1, p_2)\}$
16 $\quad$ updateSCC($\mathcal{P}$, scc, $\Delta'_{\mathcal{P}_G}$)
17 $\quad$ **return** True
18 **return** False

new SCCs are formed. The details of the algorithm are presented in (Haeupler et al., 2012). The authors of (Haeupler et al., 2012) also offer insight about the complexity of the algorithm. They show that, under a mild assumption, the incremental algorithm has the best possible complexity bound.

Incrementally maintaining $\mathcal{P}_G$ and its SCCs yields a quick way to check if a trajectory of $\mathcal{T}_G$ satisfies Φ_G (line 4 of Algorithm 1). Theorem 3.1 establishes the overall complexity of Algorithm 2.

Complexity of the Off-line Algorithm

In this section, the overall complexity of Algorithm 2 is established and we show that this is the best possible under some mild assumptions. The proofs of Theorems 3.1 and 3.5 are based on the analysis from (Haeupler et al., 2012) of incremental algorithms for cycle detection and maintenance of SCCs.

Algorithm 2 uses the soft-threshold-search algorithm presented in (Haeupler et al., 2012) to incrementally maintain SCCs. The soft-threshold-search algorithm has $O(m^{\frac{3}{2}})$ complexity and is very efficient for sparse graphs (in asymptotic sense), where m is the number of edges added to $\mathcal{T}_G$. Recall that a graph is sparse if the number of edges m is asymptotically the same as the number of nodes n, i.e. $m = O(n)$.

Theorem 3.1. *The overall execution time of the incremental search algorithm (Algorithm 2) is $O(n^{\frac{3}{2}})$, where $n = |\mathcal{T}_G|$ is the number of states added to $\mathcal{T}_G$ in Algorithm 1.*

Remark 3.2. *First, note that the execution time of the incremental procedure is better by a polynomial factor than naively running a linear-time SCC algorithm at each step, since this will have complexity $O(m^2)$, where $m = |\Delta_G|$. The algorithm presented in (Haeupler et al., 2012) improves the previously best known bound by a logarithmic factor (for sparse graphs). The proof of Theorem 3.1 exploits the fact that the "sparseness" (metric) property we defined implies a topological sparseness, i.e., $\mathcal{T}_G$ is a sparse graph.*

Proof. The soft-threshold-search algorithm attains the desired complexity only for sparse graphs. Therefore, what we need to show is that the transition system generated by Algorithm 1 is a sparse graph. Note that although we run the SCC algorithm on the product automaton, the asymptotic execution time is not affected by analyzing the transition system instead of the product automaton, because the Büchi automaton is fixed. This follows from $|S_{\mathcal{P}_G}| \leq |S_{\mathcal{B}}| \cdot |X_G|$ and $|\Delta_{\mathcal{P}_G}| \leq |\delta_{\mathcal{B}}| \cdot |\Delta_G|$.

Intuitively, the underlying graph of $\mathcal{T}_G$ is sparse, because the states were generated "far" from each other. When a new state is added to $\mathcal{T}_G$, it will be connected to other states that are at least η_1 and at most η_2 distance away. Also, all states in $\mathcal{T}_G$ are at least η_1 distance away from each other. This implies that there is a bound on the density of states. Using this intuition, the problem of estimating the maximum number of neighbors of a state can be restated as a sphere packing problem (Conway and Sloane, 1999).

Let x be the state added to $\mathcal{T}_G$ and S_1 and S_2 be two spheres centered at x and with radii η_1 and η_2, respectively. Each neighbor of x can be thought of as a sphere with radius $\eta_1/2$ and center belonging to the volume delimited by the two spheres S_1 and S_2. Since, $\eta_1 < \eta_2 < c\eta_1$, for some $c > 1$, it follows that there will be only a finite number of spheres which can be placed inside the described volume. Let N_S be the number of spheres, then a conservative upper bound is given by the following ratio

$$N_S \leq \frac{V(\eta_2) - V(\eta_1)}{V(\frac{\eta_2}{2})} < \frac{V(c\eta_1) - V(\eta_1)}{V(\frac{\eta_1}{2})} = 2^d(c^d - 1) \leq 2^{d(1+\log_2 c)} \tag{3.2}$$

where d is the dimension of the configuration space $\mathcal{C}$ and $V(\alpha)$ is the volume of a d-sphere of radius $\alpha \geq 0$. Thus, x has at most $O(1)$ neighbors. This implies that Algorithm 1 adds at most $O(1)$ transitions to $\mathcal{T}_G$ when adding a new state x. Since $\mathcal{T}_G$ is a sparse graph before adding the state x, it follows that $\mathcal{T}_G$ will remain a sparse graph. □

Remarks 3.3. *Note that the exact value of N_S may depend not only on the dimension d of the configuration space $\mathcal{C}$, but also on the shape of $\mathcal{C}$ if x is close to its boundary.*

N_S is closely related to the kissing number (Conway and Sloane, 1999) in dimension d. The kissing number τ_d is the maximum number of non-overlapping d-spheres that touch another given d-sphere. It is easy to see that τ_d is a lower bound for the

maximum value of N_S. In (Conway and Sloane, 1999), a linear optimization procedure to compute an upper bound for any dimension is presented. It is also known (Talata, 1998) that τ_d is exponential in d, i.e. $\tau_d \geq 2^{\alpha d}$, where $\alpha > 0$ is a constant. Thus, the maximum value of N_S is of order 2^d.

In (Haeupler et al., 2012), Haeupler et.al. show that any incremental algorithm that maintain a topological order and satisfies a "locality" property must take at least $\Omega(n\sqrt{m})$ time, where n is the number of nodes in the graph and m is the number of edges. The "locality" property is a mild assumption that restricts the algorithm to reorder only vertices that are affected by the addition of an edge. A vertex x is affected by the additional edge u, v if there is another vertex y such that $x < y$ in the original topological ordering, but must be changed to $x > y$. For more details see (Haeupler et al., 2012). However, it is conjectured (Haeupler et al., 2012) that this bound holds in general (Conjecture 3.4).

In the following, we assume that $m = \Omega(n)$. Also, to simplify the exposition, we assume without loss of generality that initially $\mathcal{T}_G$ has all n states and no transitions. This assumption is not restrictive, because vertex addition takes only $O(1)$.

Conjecture 3.4. *Any incremental cycle detection algorithm takes at least $\Omega(n\sqrt{m})$ time, where n is the number of vertices the graph and m is the number of edges added to it.*

Theorem 3.5. *If Conjecture 3.4 is true, then the complexity of any incremental checking algorithm for satisfying paths in a given transition system $\mathcal{T}_G$ is at least $\Omega(n\sqrt{m})$, where $n = |\mathcal{T}_G|$ and m is the number of transitions added to $\mathcal{T}_G$.*

Proof. Let $\mathcal{T}_G$ be a transition system with n states and m transitions, and $\Delta_{\mathcal{T}_G} = \{tr_1, \ldots, tr_m\}$. In the following we consider algorithms that return true or false whether adding a given transition to a transition system $\mathcal{T}_G$ yields a satisfying run

or not with respect to a given specification Φ_G. Let $A(\mathcal{T}_G, \Phi_G)$ be an incremental checking algorithm. We want to show that any such incremental algorithm takes at least $\Omega(n\sqrt{m})$ time.

It is well known (Baier and Katoen, 2008) that for any ω-regular language L there is a corresponding non-deterministic Büchi automaton, which accepts all and only the (infinite) words of L. As such, any encoding of the specification (LTL, CTL, CTL*, μ-calculus, etc.) has a corresponding Büchi automaton. Let $\mathcal{B}$ be the Büchi automaton corresponding to the ω-regular specification and $\bar{\mathcal{P}}_G = \mathcal{T}_G \times \mathcal{B}$ be the full product automaton without the reachability requirement.

Assume without loss of generality that the first $m-1$ transitions of $\mathcal{T}_G$ do not induce a satisfying run. Thus, only the mth transition may induce a satisfying run. Note, that the assumption is not limiting, because after a satisfying run is detected any additional transition will not change the result.

Let $\mathcal{P}_G^0, \ldots, \mathcal{P}_G^m$ be a sequence of subgraphs of $\bar{\mathcal{P}}_G$ with the following properties:

1. $\mathcal{P}_G^0, \ldots, \mathcal{P}_G^{m-1}$ are acyclic;

2. $\mathcal{P}_G^m$ is cyclic if and only if there is a satisfying run in $\mathcal{T}_G$ with respect to Φ_G;

3. $\emptyset = \Delta_{\mathcal{P}_G^0} \subseteq \Delta_{\mathcal{P}_G^1} \subseteq \ldots \subseteq \Delta_{\mathcal{P}_G^m}$;

4. $m' = \left|\Delta_{\mathcal{P}_G^m}\right| = \Omega(m)$.

It follows that procedure A solves the incremental cycle detection problem for $\mathcal{P}_G^0, \ldots, \mathcal{P}_G^m$. Therefore, A must take at least $\Omega(n'\sqrt{m'}) = \Omega(n\sqrt{m})$.

To complete the proof, we must show that there exists a subsequence $(\mathcal{P}_G^i)_{0 \leq i \leq m}$ for a given $\bar{\mathcal{P}}_G$ and a sequence $\{tr_1, \ldots tr_m\}$ of transitions of $\mathcal{T}_G$. We will define the subgraphs recursively as follows: (1) $\mathcal{P}_G^m$ is the maximum acyclic spanning subgraph of $\bar{\mathcal{P}}_G$ if $\mathcal{T}_G$ does not contain a satisfying run or $\mathcal{P}_G^m = \bar{\mathcal{P}}_G$, otherwise; (2) $\mathcal{P}_G^i$ is

the maximum acyclic spanning subgraph of $\mathcal{P}_G^{i+1}|_{E_i}$ for all $i \in \{0, \ldots, m-1\}$, where $E_i = \{tr_1, \ldots, tr_i\} \times \delta_{\mathcal{B}}$ and $\mathcal{P}_G^{i+1}|_{E_i}$ is the subgraph of $\mathcal{P}_G^{i+1}$ with transitions restricted to E_i. From the definition it immediately follows that $\Delta_{\mathcal{P}_G^i} \subseteq (E_i \cap \Delta_{\mathcal{P}_G^{i+1}}) \subseteq \Delta_{\mathcal{P}_G^{i+1}}$ for all $0 \leq i \leq m-1$ and $\Delta_{\mathcal{P}_G^0} = \emptyset$. Thus, by construction conditions (1), (2) and (3) are satisfied. The last requirement is trivially true when $\mathcal{T}_G$ contains a satisfying run. Also, when $\mathcal{T}_G$ does not contain a satisfying run, then the maximum acyclic graph of $\bar{\mathcal{P}}_G$ retains at least half the transitions. Any digraph G may be decomposed into two acyclic subgraphs (Wood, 2004) such that their edge sets form a partition of the edge set of G. It follows that at least one (acyclic) subgraph has half of the edges of G. Thus, we have that $m' = \Omega(m)$. □

Remark 3.6. *Note that Theorem 3.1 gives a lower bound for all incremental checking procedures with respect to the number of states and transitions which are added.*

The following corollaries of Theorem 3.5 are easy to prove.

Corollary 3.7. *If Conjecture 3.4 is true, then Algorithm 2 has the best possible complexity for transition systems that are sparse graphs.*

Corollary 3.8. *Algorithm 2 has the best possible complexity for transition systems, which are sparse graphs, among all incremental algorithms with the "locality" property.*

Probabilistic completeness

The presented RRG-based algorithm retains the probabilistic completeness of RRT, since the constructed transition system is composed of an RRT-like tree and some transitions that close cycles.

Theorem 3.9. *Algorithm 1 is probabilistically complete.*

Proof. First we start by noting that any word in a ω-regular language can be generated by a finite prefix path and a finite suffix path that is repeated indefinitely in the corresponding Büchi automaton (Baier and Katoen, 2008). This is important, since this shows that a solution, represented by a transition system, is completely characterized by a finite number of states. Let us denote by $\bar{X}$ the finite set of states that define a solution. It follows from the way regions are defined that we can choose a neighborhood around each state in $\bar{X}$ such that the system can be steered in one step from all points in one neighborhood to all points in the next neighborhood. Thus, we can use induction to show that (Karaman and Frazzoli, 2012): (1) there is a non-zero probability that a sample will be generated inside the neighborhood of the first state in the solution sequence; (2) if there is a state in $\mathcal{T}$ that is inside the neighborhood of the k-th state from the solution sequence, then there is a non-zero probability that a sample will be generated inside the $k+1$-st state's neighborhood. Therefore, as the number of samples goes to infinity, the probability that the transition system $\mathcal{T}$ has nodes belonging to all neighborhoods of states in $\bar{X}$ goes to 1. To finish the proof, note that we have to show that the algorithm is always able to generate samples with the desired "sparseness" property. However, recall that the bound functions must converge to 0 (as the number of states goes to infinity) fast enough such that the set of configurations for which "far" function returns a non-empty list has non-zero measure with respect to the sampling distribution. This concludes the proof. □

3.2.2 On-line algorithm

The approach for solving the on-line part of the planning problem is based on the RRT algorithm, a probabilistically complete sampling-based path planning method. We modify the standard RRT in order to find local paths which preserve the satisfaction of the global specification Φ_G, while servicing on-line requests and avoiding locally

sensed obstacles.

To keep track of validity of samples (random configurations) with respect to the global specification Φ_G, we propose a method that combines the ideas presented in (Bauer et al., 2011) on monitors for LTL formulae and (Ding et al., 2014) on potential functions. The problem considered in (Bauer et al., 2011) is to decide as soon as possible if a given (infinite) word w satisfies a LTL formula ϕ. The main idea is to keep track of Büchi states corresponding to a finite prefix of w with respect to both ϕ and $\neg\phi$ concurrently. If one of the two sets of Büchi states corresponding to ϕ or $\neg\phi$ becomes empty, then we can conclude that the specification is either violated or satisfied. If both sets are non-empty then nothing can be said about $w \models \phi$. In our case, we just use half of a monitor, since we are interested only in checking if steering the robot to new samples violates Φ_G. The potential functions approach described in (Ding et al., 2014) is used to address the problem of connecting the locally generated path to states in the global transition system such that Φ_G is satisfied.

Potential functions

In (Ding et al., 2014) the authors define a potential function over the states of the product automaton between a transition system and a Büchi automaton. The potential function captures the distance from each state of the product to the closest final state. It can be thought of as a distance to satisfaction and resembles a Lyapunov function. We extend this notion to define potential functions on the states of the global transition system. This extension allows us to reason about the change of potential between nodes of $\mathcal{T}_G$ connected through local paths instead of a direct transition. The local paths are generated as branches of a tree by the proposed RRT-based algorithm. The definitions of self-reachable set and potential function for

product automaton states presented below are adapted from (Ding et al., 2014).

Let $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B} = (S_{\mathcal{P}_G}, S_{\mathcal{P}_{G0}}, \Delta_{\mathcal{P}_G}, \omega_{\mathcal{P}_G}, F_{\mathcal{P}_G})$ be a product automaton between a transition system $\mathcal{T}_G$ and Büchi automaton $\mathcal{B}$. We denote by $\mathcal{D}(p, p')$ the set of all finite trajectories from a state $p \in S_{\mathcal{P}_G}$ to a state $p' \in S_{\mathcal{P}_G}$:

$$\mathcal{D}(p, p') = \{p_1 \dots p_n | p_1 = p, p_n = p'; p_k \rightarrow_{\mathcal{P}_G} p_{k+1} \forall k = 1, \dots, n-1; \forall n \geq 2\} \quad (3.3)$$

A state $p \in S_{\mathcal{P}_G}$ is said to reach a state $p' \in S_{\mathcal{P}_G}$ if $\mathcal{D}(p, p') \neq \emptyset$. The length of a path is defined as the sum of the weights corresponding to the transitions it is composed of:

$$L(\mathbf{p}) = \sum_{k=1}^{n-1} \omega_{\mathcal{P}_G}(p_k, p_{k+1}) \quad (3.4)$$

For $p, p' \in S_{\mathcal{P}_G}$, the distance between p and p' is defined as follows:

$$d(p, p') = \begin{cases} \min_{\mathbf{p} \in \mathcal{D}(p,p')}(L(\mathbf{p})) & \text{if } \mathcal{D}(p, p') \neq \emptyset \\ \infty & \text{if } \mathcal{D}(p, p') = \emptyset \end{cases} \quad (3.5)$$

The weight function $\omega_{\mathcal{P}_G}$ is positive, because it is induced by the distance of the underlying (metric) space. This implies (Ding et al., 2014) that $d(p, p') > 0$ for all $p, p' \in S_{\mathcal{P}_G}$.

A set $A \subset S_{\mathcal{P}_G}$ is *self-reachable* if and only if all states in A can reach a state in A. Formally, a set A is self-reachable if for all $p \in A$ there is a state p' such that $\mathcal{D}(p, p') \neq \emptyset$.

Definition 3.1 (Potential function of states in $\mathcal{P}_G$)**.** *The potential function $V_{\mathcal{P}_G}(p)$, $p \in S_{\mathcal{P}_G}$ is defined as:*

$$V_{\mathcal{P}_G}(p) = \begin{cases} \min_{p' \in F^*_{\mathcal{P}_G}} d(p, p') & \textit{if } p \notin F^*_{\mathcal{P}_G} \\ 0 & \textit{if } p \in F^*_{\mathcal{P}_G} \end{cases} \quad (3.6)$$

where $F^*_{\mathcal{P}_G} \subset F_{\mathcal{P}_G}$ is the maximal self-reachable set of final states of $\mathcal{P}_G$.

The potential function is non-negative for all states of $\mathcal{P}_G$. It is zero for some $p \in S_{\mathcal{P}_G}$ if and only if p is a final state and p can reach itself or a self-reachable final state. Also, if $V_{\mathcal{P}_G}(p) = \infty$, $p \in S_{\mathcal{P}_G}$, then p does not reach any self-reachable final states.

Definition 3.2 (Potential function of states in $\mathcal{T}_G$)**.** *Let $x \in X$ and $B \subseteq \beta_{\mathcal{P}_G}(x)$. The potential function of x with respect to B is defined as:*

$$V_{\mathcal{T}_G}(x, B) = min_{s \in B} V_{\mathcal{P}_G}((x, s)) \tag{3.7}$$

Also, the minimum potential of x is defined as $V^*_{\mathcal{T}_G}(x) = V_{\mathcal{T}_G}(x, \beta_{\mathcal{P}_G}(x))$.

The minimum potential of a state x of $\mathcal{T}_G$ is the minimum potential of all states in $\mathcal{P}_G$ which correspond to x. The actual potential is defined to capture the fact that not all Büchi states may be available in order to achieve the minimum potential.

In (Ding et al., 2014), the authors present an algorithm to compute the potential function $V_{\mathcal{P}_G}(\cdot)$ over the states of the product automaton. The complexity of the algorithm is $O(|F_{\mathcal{P}_G}|^3 + |F_{\mathcal{P}_G}|^2 + |S_{\mathcal{P}_G}|^2 \times |F_{\mathcal{P}_G}|)$ (Ding et al., 2014).

We propose an improved algorithm (see Algorithm 3), which reduces the complexity by a polynomial factor.

Theorem 3.10. *Algorithm 3 correctly computes the potential function $V_{\mathcal{P}_G}(\cdot)$ for a given product automaton $\mathcal{P}_G$ and its complexity is $O(|S_{\mathcal{P}_G}| \log |S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$.*

Remark 3.11. *In the proposed framework, the computation of the SCC in Algorithm 3 (lines 1–3) may be skipped, because the off-line planning Algorithm 1 already maintains SCCs of $\mathcal{P}_G$. Thus, Algorithm 3 is better suited for use in conjunction with the off-line algorithms.*

Algorithm 3: Compute potential function $V_{\mathcal{P}_G}(\cdot)$

Input: $\mathcal{P}_G$ – product automaton
Output: Boolean value indicating whether there are self-reachable final states

1 $\mathcal{P}_G \leftarrow \mathcal{P}_G \cup (\{v\}, \{(v, p, 0) : p \in S_{\mathcal{P}_G}\})$ `// add virtual state` v `and connect it to all initial states`
2 $scc, dag \leftarrow StronglyConnectedComponentsDAG(\mathcal{P}_G)$ `// compute SCC DAG for` $\mathcal{P}_G$
3 $scc_v \leftarrow \{v\}$
4 $F^*_{\mathcal{P}_G} \leftarrow ComputeSRFS(dag, F_{\mathcal{P}_G}, scc_v)$ `// compute self-reachable final states`
5 $\mathcal{P}_G \leftarrow \mathcal{P}_G \setminus \{v\}$ `// remove virtual state` v `and all its incident transitions`
6 **if** $F^*_{\mathcal{P}_G} = \emptyset$ **then** `// if there are no self-reachable final states`
7 ⌊ **return** False
8 $\mathcal{P}_G \leftarrow \mathcal{P}_G \cup (\{v\}, \{(p, v, 0) : p \in F^*_{\mathcal{P}_G}\})$ `// add virtual state` v `and connect all self-reachable final states to it with weight 0`
9 $V_{\mathcal{P}_G} \leftarrow ReverseDijkstra(\mathcal{P}_G, sink = v)$ `// compute potentials for each state with` v `as sink`
10 $\mathcal{P}_G \leftarrow \mathcal{P}_G \setminus \{v\}$ `// remove virtual state` v `and all its incident transitions`
11 **return** True

Proof. The improvement achieved by Algorithm 3 is based on two observations: (1) if the maximal self-reachable final states set $F^*_{\mathcal{P}_G}$ is known then the potential function $V_{\mathcal{P}_G}(\cdot)$ can be computed by running Dijkstra's algorithm once instead of $|F_{\mathcal{P}_G}|$ times as in (Ding et al., 2014); (2) self-reachability is a property about the existence of cycles in $\mathcal{P}_G$ and can therefore be inferred from the SCC directed acyclic graph (DAG) of $\mathcal{P}_G$.

Algorithm 3 computes the potential function $V_{\mathcal{P}_G}(\cdot)$ by first computing $F^*_{\mathcal{P}_G}$ (lines 1–5) using the SCC DAG (line 2) and Algorithm 4. However, Algorithm 4 performs a depth-first search (DFS) of *dag* starting from a given SCC scc_r. Thus, it returns only the states of $F^*_{\mathcal{P}_G}$ which belong to scc_r and its descendants. In order to avoid calling Algorithm 4 for all SCC, we add a virtual node v to $\mathcal{P}_G$ (line 1), which is connected to

Algorithm 4: Compute Self-Reachable Final States – computeSRFS()

Input: dag – the SCC directed acyclic graph of $\mathcal{P}_G$
Input: $F_{\mathcal{P}_G}$ – the set of final states of $\mathcal{P}_G$
Input: scc_r – the current root SCC used by the DFS algorithm
Output: $srfs$ – the set of self-reachable final states

```
1  srfs ← ∅
2  visited(scc_r) ← True
3  foreach scc_n ∈ dag.out(scc_r) do
4      if ¬visited(scc_n) then
5          srfs ← srfs ∪ computeSRFS(dag, F_{P_G}, scc_n)
6      else
7          srfs ← srfs ∪ S(scc_n)
8  if |scc_r| > 1 ∨ srs ≠ ∅ ∨ (scc_r = {p} ∧ (p, p) ∈ Δ_{P_G}) then
9      srfs ← srfs ∪ (F_{P_G} ∩ scc_r)
10 S(scc_r) ← srfs
11 return srfs
```

all states of $\mathcal{P}_G$, and then compute the SCC DAG. Because v only has outgoing transitions, it can not belong to any cycle. Thus, the SCC scc_v containing v is a singleton and is connected to all other SCCs in dag. It follows that running Algorithm 4 on dag with starting SCC scc_v (line 4) correctly computes $F^*_{\mathcal{P}_G}$. Afterwards, v is removed and all incident transitions from $\mathcal{P}_G$ (line 5). If there are self-reachable final states (line 6), then the algorithm proceeds to compute the potentials using Dijkstra's algorithm starting from $F^*_{\mathcal{P}_G}$ and traversing transitions in the opposite direction (lines 8–10), i.e. using the incoming transitions instead of the outgoing transitions. Again, in order to avoid calling Dijkstra's algorithm for every state in $F^*_{\mathcal{P}_G}$, we add a virtual node v to $\mathcal{P}_G$. All states in $F^*_{\mathcal{P}_G}$ are connected to v with weight 0. Because v has only ingoing transitions and $\omega_{\mathcal{P}_G}((p, p')) > 0$ for all $(p, p') \in \Delta_{\mathcal{P}_G}$, it follows that v does not belong to any cycles and Dijkstra's algorithm correctly computes the potential function for

every $p \in S_{\mathcal{P}_G}$:

$$
\begin{aligned}
d(p,v) &= \begin{cases} \min_{p' \in F^*_{\mathcal{P}_G}} \{d(p,p') + \omega_{\mathcal{P}_G}((p',v))\} & \text{if } p \notin F^*_{\mathcal{P}_G} \\ \omega_{\mathcal{P}_G}((p',v)) & \text{if } p \in F^*_{\mathcal{P}_G} \end{cases} \\
&= \begin{cases} \min_{p' \in F^*_{\mathcal{P}_G}} d(p,p') & \text{if } p \notin F^*_{\mathcal{P}_G} \\ 0 & \text{if } p \in F^*_{\mathcal{P}_G} \end{cases} \\
&= V_{\mathcal{P}_G}(p)
\end{aligned}
$$

The analysis presented above relies on the fact that Algorithm 4 correctly computes $F^*_{\mathcal{P}_G}$. In the following, we prove by structural induction with respect to dag that Algorithm 4 correctly computes $F^*(scc_r)$, where scc_r is an SCC of dag and $F^*(scc_r)$ is the maximal subset of $F^*_{\mathcal{P}_G}$ whose states belong to scc_r and its descendants.

First, note that by definition a final state p_f belongs to $F^*_{\mathcal{P}_G}$ if and only if: (1) p_f belongs to a cycle or equivalently to a SCC of $\mathcal{P}_G$ with more than one state; (2) p_f has a self-loop; or (3) p_f reaches another state in $F^*_{\mathcal{P}_G}$. Since dag is acyclic if follows that condition (3) can be reduced to checking if $F^*(scc_n)$ is non-empty for some successor scc_n of scc_r. This implies that $F^*(scc_r)$ is unique for every scc_r and it can be computed recursively using depth-first search. The recursive algorithm starts by marking the current SCC scc_r as visited (line 2) and proceeds to compute the union of $F^*()$ for all successors of scc_r (lines 3–7). If a successor scc_n was not previously visited then the procedure is called recursively starting from scc_n (lines 4–5), otherwise the stored set corresponding to scc_n is used (line 7). The next step is to add the self-reachable final states of scc_r to $srfs$ (lines 8–9). The $srfs$ is stored in $S(scc_r)$ for possible later use. We need to show that $S(scc_r) = F^*(scc_r)$, for all scc_r in dag.

The *base case* is trivial, because it involves the SCCs without any outgoing transitions in dag. It follows that $srfs$ at line 8 is empty. Also, all final states in scc_r satisfying conditions (1) or (2) are added to $srfs$. Thus, $S(scc_r) = F^*(scc_r)$.

For the *induction step*, we assume that Algorithm 4 correctly computes $F^*(scc_n)$ for all successors of scc_r (line 5). Note that if a successor scc_n was already visited at some previous step, Algorithm 4 was called with scc_n as starting SCC. Therefore, $S(scc_n)$ (line 7) is assumed to be computed correctly by the induction hypothesis. As in the base case, if either condition (1) or (2) hold, then Algorithm 4 adds all final states in scc_r to srs and it follows that $S(scc_r) = F^*(scc_r)$. The remaining case is when scc_r is a singleton $\{p\}$ and p has no self-loop. In this case, $p \in F^*(scc_r)$ if and only if p reaches some other state in $F^*(scc_r)$. Since dag is acyclic, p can only reach states in the descendants SCC of scc_r. On the other hand, by the induction hypothesis we have that $srfs = F^*(scc_r) \setminus \{p\}$ at line 8. It follows that p is added to $S(scc_r)$ if $srfs$ is non-empty at line 8. Thus, we have $S(scc_r) = F^*(scc_r)$ in this case as well when Algorithm 4 returns.

The complexity of Algorithm 3 is $O(|S_{\mathcal{P}_G}| \log |S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$. It is easy to see that the operations on lines 1, 5, 8 and 10 take $O(|S_{\mathcal{P}_G}|)$, while computing the SCC DAG (line 2) and Dijkstra's algorithm (line 9) have $O(|\Delta_{\mathcal{P}_G}|)$ and $O(|S_{\mathcal{P}_G}| \log |S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$ complexity, respectively. Also, computing $F^*_{\mathcal{P}_G}$ using Algorithm 4 takes at most $O(|S_{\mathcal{P}_G}| + |\Delta_{\mathcal{P}_G}|)$. The SCC DAG graph dag has at most the same number of states and transitions as $\mathcal{P}_G$. Also, Algorithm 4 is a DFS and each SCC is processed once and each transition of dag is transversed once. Therefore, the overall complexity of processing the SCCs in Algorithm 4 (lines 8–10) is linear in the number of states of $\mathcal{P}_G$. Adding the complexity of all steps, we obtained the stated complexity bound. □

Satisfying local paths with respect to Φ_G

Local paths in our RRT based algorithm connect states of the global transition system. Let $x, x' \in \mathcal{T}_G$ and $\mathbf{x} = x_1 \dots x_n$ be a local path connecting $x_1 = x$ and $x_n = x'$ and $\mathbf{o} = o_1 \dots o_n$ be the output trajectory corresponding to $\mathbf{x}$ with respect to the global proprieties ($o_k \in 2^{\Pi_G}, \forall k = 1 \dots n$). We need to ensure that there is a satisfying run in $\mathcal{T}_G$ starting at x' after traversing $\mathbf{x}$. Thus, we need to consider two problems: (1) how to keep track of available Büchi states as local samples are generated and (2) how to connect a local path's endpoint (tree leaf) to the global transition system $\mathcal{T}_G$.

The first problem is solved by Algorithm 5, which determines the set of Büchi states given a word w over 2^{Π_G}. Algorithm 5 solves this problem by repeatedly computing the set of outgoing neighboring states of $\mathcal{B}$ for all states in the previous iteration. To check if a local path can be connected to the state $x_n = x' \in X$, we just need to verify that it has finite potential, i.e. $V_{\mathcal{T}_G}(x', B) < \infty$, where B is the set of available Büchi states after traversing $\mathbf{x}$, in this case $w = \mathbf{o}$.

The second problem has a simple solution in this setting. We choose the state in $\mathcal{T}_G$ which has (finite) minimum potential after traversing a branch of the RRT tree. Also, the line segment between the leaf state from the tree and the state in $\mathcal{T}_G$ must be collision free (see Section 7).

On-line planning algorithm

The overall planning algorithm, outlined in Algorithm 6, is composed of the off-line preprocessing steps of computing the global transition system $\mathcal{T}_G$, the product automaton $\mathcal{P}_G = \mathcal{T}_G \times \mathcal{B}$ and the potential function for $\mathcal{P}_G$ and the on-line loop. At each step of the loop, the robot scans for local requests and obstacles and checks if it needs to compute a new local path. Re-planning is performed in four cases: (1) if the current path is empty; (2) a higher priority request was detected; (3) the chosen

Algorithm 5: Tracking Büchi states of local samples

Input: $\mathcal{B}$ – Büchi automaton corresponding to Φ_G
Input: $w = \sigma_1 \dots \sigma_n$ – a finite word over 2^{Π_G}
Input: B – a set of Büchi states from which the tracking starts
Output: B_f – set of Büchi states available after the last symbol of w

```
1 B_f ← B
2 for k ← 1 ... (n − 1) do
3     B' ← ∅
4     foreach s ∈ B_f do
5         B' ← B' ∪ {s' ∈ S_B | (s, s') ∈ Δ_B}
6     B_f ← B'
7 return B_f
```

request disappeared; and (4) the local path collides with a local obstacle. Büchi states are tracked starting from the initial configuration of the robot, corresponding to the initial state of $\mathcal{T}_G$. Map B is used to store the tracked Büchi states. Figure 3·3 shows how a the local planning algorithm interacts with the $\mathcal{T}_G$ and locally sensed requests and obstacles.

The local path planning algorithm is shown in Algorithm 7 and is based on RRT. The procedure incrementally constructs the local transition system $\mathcal{T}_L$. The initial (root) state of $\mathcal{T}_L$ is the current configuration of the robot x_c. The map *serv* indicates whether a state or any of its ancestors serviced the on-line request with the highest priority. If there are no requests then *serv* is true for all states of $\mathcal{T}_L$.

The construction of the RRT proceeds by generating a new random sample (line 4) inside the sensing area of the robot, steer the system towards it (lines 5–6) and checking if it is a valid state (lines 8–9). Samples are generated such that their images in $\mathcal{D}$ belong to the sensing area of the robot. The *nearest* function (line 5) is a standard RRT primitive which returns the nearest state in $\mathcal{T}_L$ based on the distance function associated with $\mathcal{C}$. We assume that we have access to a *steer* function (see Section 3.2.1) which drives the system from x_n to a configuration $x \in$

Algorithm 6: Planning algorithm

Input: Φ_G – the global LTL$_{-X}$ specification
Input: *prio* – the priority function for on-line requests
Input: x_0 initial configuration of the robot

```
1  Convert Φ_G to Büchi automaton B
2  Compute T_G and P_G = T_G × B starting at x_0 using Algorithm 1
3  Compute potential function V_{P_G}(·)
4  path ← emptyList()
5  x_c ← x_0
6  B(x_c) ← β_{P_G}(x_c)
7  while True do
8      I ← getLocalRequests()
9      if checkPath(I, path) ∨ ¬path.hasNext() then
10         path ← planLocally(x_c, P_G, B, prior, I)
11     x_n ← path.next()
12     enforce(x_c → x_n)
13     x_c ← x_n
```

$\mathcal{C}$, where x is the closest configuration to the new sample x_s and it is within η_L distance from x_n (LaValle and Kuffner, 1999; Karaman and Frazzoli, 2011b). The *label* primitive function (line 7) is used to annotate x with the global properties it satisfies. The new state x is valid if its corresponding set of Büchi states is non-empty and the line segment from its parent x_n to itself is a simple collision free line segment. Algorithm 5 is used to compute the set of available Büchi states for x. The primitive function *isSimpleSegment* is used to ensure that the set of global properties along the potential new transition (x_n, x) changes at most once (see Section 3.2.1). The *collisionFree* primitive is used to check if the image in the workspace of the line segment $(x_n, x) \in \mathcal{C}$ collides with a local obstacle in $\mathcal{D}$. If these tests are passed, the procedure adds the state x and the transition (x_n, x) to $\mathcal{T}_L$ (line 11–12). Also the *serv* map is updated by checking if either the parent state x_n (or some ancestor) or the state itself x has serviced the selected on-line request.

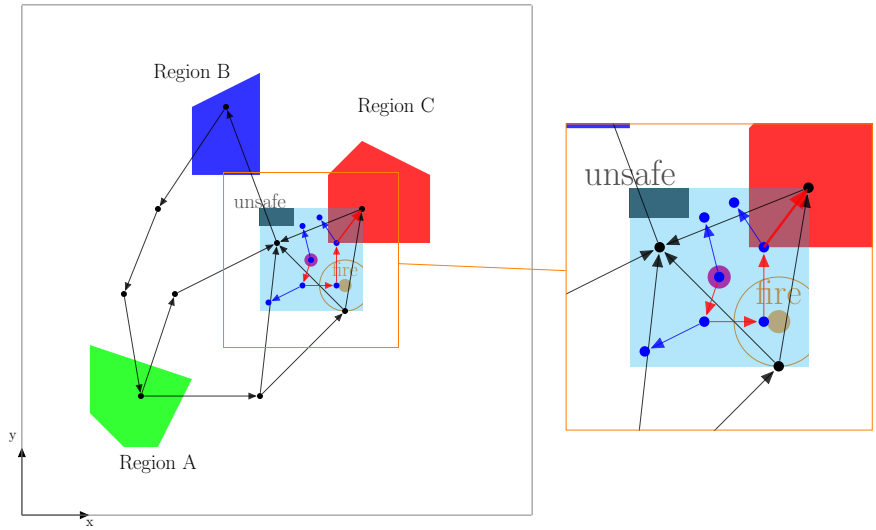


Figure 3·3: The figure presents the same environment as in Figure 3·1, but also shows the global transition system $\mathcal{T}_G$ (in black) and the local transition system $\mathcal{T}_L$ (in blue and red). The robot's current position x_c is marked by the magenta disk and coincides with the root of $\mathcal{T}_L$. The sensing area is again in cyan and a $fire$ request and a local obstacle ($unsafe$) are detected. Note that in this figure only the portion of the $unsafe$ area which is inside the sensing area is detected. Also, the *survivor* request is not detected at all. The local control strategy, which corresponds to a path from x_c to a leaf and then to a state in $\mathcal{T}_G$, was found and is shown in red. The last transition of the local path is the link between $\mathcal{T}_L$ and $\mathcal{T}_G$. This local path satisfies the global and local mission specification described in Example 3.1.

Also, we require that the state x_G of $\mathcal{T}_G$ have a lower (actual) potential than the last visited state x'_G of $\mathcal{T}_G$. This condition is not enforced, if the potential of x'_G is zero, but we still require $x_G \neq x'_G$.

Correctness of local paths with respect to Φ_G

Theorem 3.12. *Let* $\mathbf{x} = x_1, \ldots$ *be an infinite path in* $\mathcal{C}$ *generated by Algorithm 6 and* $\mathbf{o} = o_1, \ldots$ *be the corresponding (infinite) output word generated by traversing* $\mathbf{x}$*. If every call of Algorithm 7 finishes in finite time, then* $\mathbf{o}$ *satisfies the global specification* Φ_G*, i.e.,* $\mathbf{o} \models \Phi_G$*.*

Algorithm 7: Local path planning

Input: x_c – current configuration of the robot
Input: $\mathcal{P}_G$ – the product automaton $\mathcal{T}_G \times \mathcal{B}$
Input: $\mathcal{B}$ – Büchi automaton corresponding to global specifications Φ_G
Input: *prior* – on-line requests priority function
Input: I – sensed requests and local obstacles
Output: *path* – computed local control strategy

```
1  Construct T_L = (X_L, x_c, Δ_L, ω_L, Π_L ∪ {π_O}, h_L) with x_c as initial state
2  serv(x_c) ← ¬I.hasRequest()
3  while ∄ x_c →*_{T_L} x_T → x_G w/ V_{T_G}(x_G, B(x_G)) < ∞ ∨ ¬serv(x_T) do
4      x_s ← generateSample(x_c, I.area)
5      x_n ← nearest(T_L, x_s)
6      x ← steer(x_n, x_s, η_L)
7      x ← label(x, I)
8      B(x) ← trackBuchiStates(B, h_L(x_n), B(x_n))
9      if B(x) ≠ ∅ ∧ isSimpleSegmnent(x_n, x) ∧ collisionFree(x_n, x) then
10         serv(x) ← serv(x_n) ∨ I.serviced(x, prior)
11         X_L ← X_L ∪ {x}
12         Δ_L ← Δ_L ∪ {(x_n, x)}
13 return x_c →*_{T_L} x_T → x_G
```

Proof. The condition that local path planning algorithm (Algorithm 7) always finishes in finite time implies that it was able to successfully find a local strategy every time the robot detected on-line requests and local obstacles. Therefore, this assumption implies that the environment is not adversary to the robot, i.e. it does not actively try to stop the robot from performing its mission.

By construction, every time Algorithm 7 finishes successfully it returns a local path which ends in a state x of $\mathcal{T}_G$ with finite (actual) potential. This implies that there is a state $p = (x, s)$ of $\mathcal{P}_G$ with finite potential, where $s \in B(x)$, and its potential is less than the potential of the previous state of $\mathcal{T}_G$ occurring in $\mathbf{x}$. As shown in (Ding et al., 2014), this guarantees that there is a state x' in $\mathbf{x}$ with zero potential and x' is a finite number of steps after x in $\mathbf{x}$.

By the hypothesis, $\mathbf{x}$ contains infinitely many states of $\mathcal{T}_G$ and an infinite number

of them has zero potential. This concludes the proof, since the states with zero potential correspond to final Büchi states. □

Remark 3.13. *The complexity of the local path planning algorithm (Algorithm 7) is the same as for the standard RRT. The functions generateSample, steer and nearest are stardard primitives (LaValle and Kuffner, 1999; Karaman and Frazzoli, 2011b). label, isSimpleSegment and collisionFree primitives and checking if an on-line request was serviced, can be reduced to collision detection in the lower dimensional workspace. Tracking Büchi states takes constant time ($O(1)$), because the global specification Φ_G is fixed.*

3.3 Case study

In this section, we present some examples scenarios and show that the proposed framework is able to generate off-line and on-line control policies such that the global and local mission specifications are met. At the end of the sections, we present a proof-of-concept experiment, which shows a differential drive robot performing a persistent surveillance mission in a planar environment while reacting to locally sensed events. In all cases, we assume for simplicity that the *Steer* function is trivial, i.e., there are no actuation constraints at any given configuration.

The algorithms presented in this section are implemented in Python2.7 and the *LOMAP* (Ulusoy et al., 2013c) and *networkx* (Hagberg et al., 2008) libraries. The *ltl2ba* tool (Gastin and Oddoux, 2001) was used to convert the LTL specifications into Büchi automata. All examples were ran on an iMac system with a 3.4 GHz Intel Core i7 processor and 16GB of memory.

3.3.1 Off-line algorithm

In this section, we focused on investigating the scalability of the off-line algorithm with respect to the dimension of the configuration space. As such, in the following we will assume that the workspace coincides with the configuration space and the submersion $\mathcal{H}$ is trivial.

Case Study 1: Consider the configuration space depicted in Figure 3·4. The initial configuration is at $(0.3; 0.3)$. The specification is to visit regions $r1$, $r2$, $r3$ and $r4$ infinitely many times while avoiding regions $o1$, $o2$, $o3$ and $o4$. The corresponding LTL formula for the given mission specification is

$$\phi_1 = \mathbf{G}(\mathbf{F}r1 \wedge (\mathbf{F}r2 \wedge (\mathbf{F}r3 \wedge (\mathbf{F}r4))) \wedge \neg(o1 \vee o2 \vee o3 \vee o4)) \tag{3.8}$$

A solution to this problem is shown in Figures 3·4 and 3·5. We ran the overall algorithm 20 times and obtained an average execution time of 6.954 sec, out of which the average of the incremental search algorithm was 6.438 sec. The resulting transition system had a mean size of 51 states and 277 transitions, while the corresponding product automaton had a mean size of 643 states and 7414 transitions. The Büchi automaton corresponding to ϕ_1 had 20 states and 155 transitions.

Case Study 2: Consider a 10-dimensional unit hypercube configuration space. The specification is to visit regions $r1$, $r2$, $r3$ infinitely many times, while avoiding region $o1$. The LTL formula corresponding to this specification is

$$\phi_2 = \mathbf{G}(\mathbf{F}r1 \wedge (\mathbf{F}r2 \wedge (\mathbf{F}r3)) \wedge \neg o1). \tag{3.9}$$

The corresponding Büchi automaton has 9 states and 43 transitions. Regions $r1 = [0; 0.4] \times [0; 0.75]^9$, $r2 = [0.6; 1] \times [0.25; 1]^9$, $r3 = [0.6; 1] \times [0; 0.2] \times ([0.2; 1] \times [0; 0.8])^4$ and $o1 = [0.41; 59] \times [0.3; 0.9] \times [0.12; 0.88]^8$ are hypercubes and their volumes are

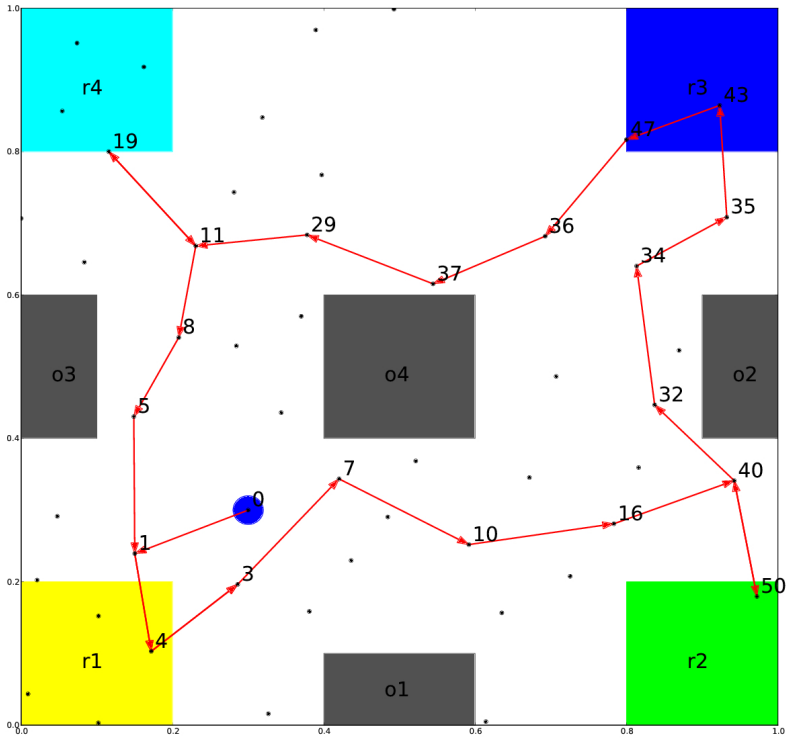


Figure 3·4: One of the solutions corresponding to Case Study 1: the specification is to visit all the colored regions labelled $r1$ (yellow), $r2$ (green), $r3$ (blue) and $r4$ (cyan) infinitely often, while avoiding the dark gray obstacles labelled $o1$, $o2$, $o3$, $o4$. The black dots represent the states of the transition system $\mathcal{T}$ (51 states and 264 transitions). The starting configuration of the robot (the initial state of $\mathcal{T}$) is denoted by the blue circle. The red arrows represent the satisfying run (finite prefix, suffix pair) found by Algorithm 1, which is composed of 21 states from $\mathcal{T}$. In this case, the prefix and suffix are [0, 1, 4, 3] and [7, 10, 16, 40, 50, 40, 32, 34, 35, 43, 47, 36, 37, 29, 11, 19, 11, 8, 5, 1, 4, 3], respectively.

0.03, 0.03, 0.013 and 0.012, respectively. $r1$, $r2$, $r3$ are positioned in the corners of the configuration space, while $o1$ is positioned in the center. In this case, the algorithm took 16.75 sec on average (20 experiments), while just the incremental search procedure for a satisfying run took 14.471 sec. The transition system had a mean size of 69 states and 1578 transitions, while the product automaton had a mean

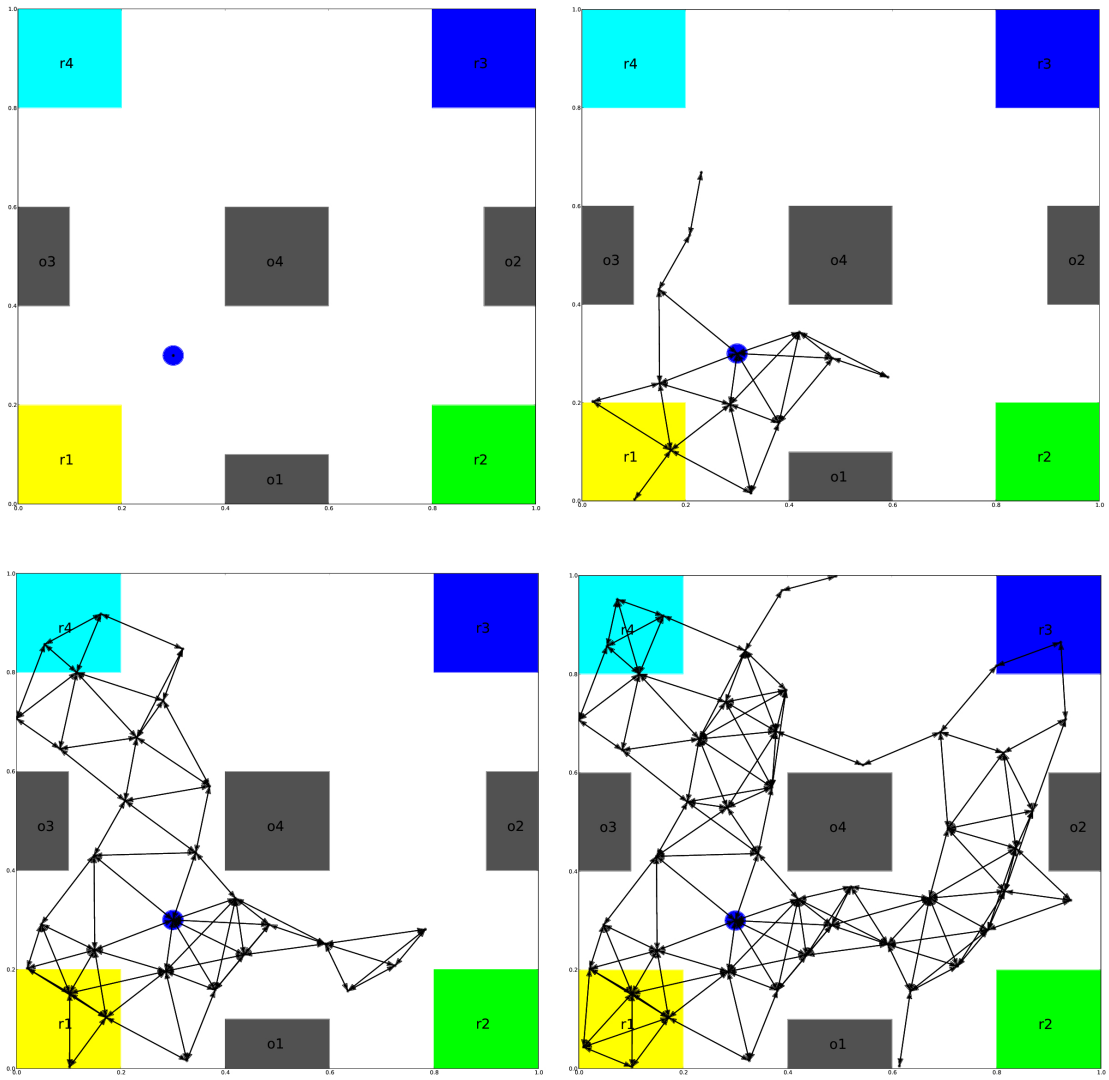


Figure 3·5: Transition systems obtained at earlier iterations corresponding to the solution shown in Figure 3·4 (to be read from left to right and top to bottom). The black dots and arrows represent the state and transitions of $\mathcal{T}$, respectively.

size of 439 states and 21300 transitions.

Case Study 3: We also considered a 20-dimensional unit hypercube configuration space. Two hypercube regions $r1$ and $r2$ were defined and the robot was required to visit both of them infinitely many times ($\phi_3 = \mathbf{G}(\mathbf{F}(r1 \wedge \mathbf{F}r2))$). The overall

algorithm took 7.45 minutes, while the transition system grew to 414 states and 75584 transitions. The corresponding product automaton had a size of 1145 states and 425544 transitions. This example illustrates the fact that the bound on the number of neighbors of a state in the transition system grows at least exponentially in the dimension of the configuration space.

3.3.2 On-line algorithm

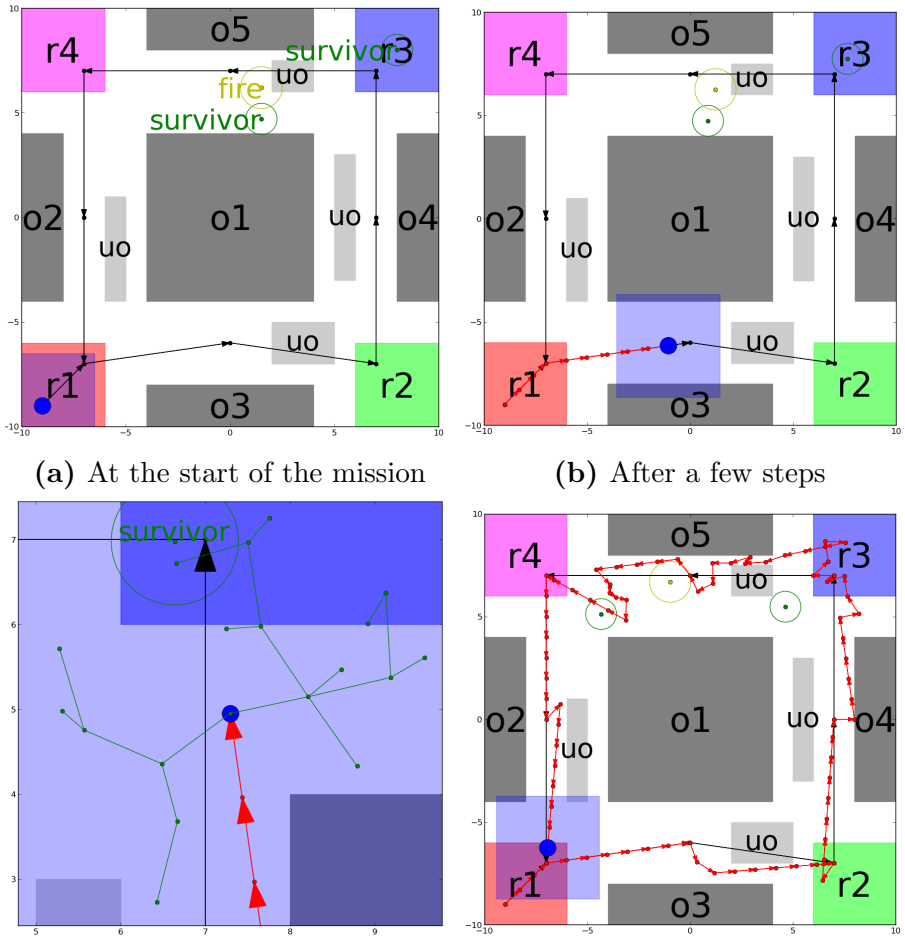
In this section, we present simplified scenario involving a fully actuated point in a planar environment to highlight the on-line planning algorithm. Thus, the workspace coincides with the configuration space and they are both two-dimensional.

Consider the configuration space depicted in Figure 3·6a. The initial configuration is $x_0 = (-9; -9)$. The global specification is to visit regions $r1$, $r2$, $r3$ and $r4$ infinitely many times while avoiding regions $o1$, $o2$, $o3$, $o4$ and $o5$. The corresponding LTL_{-X} formula is $\Phi_G = \mathbf{G}(\mathbf{F}r1 \wedge \mathbf{F}r2 \wedge \mathbf{F}r3 \wedge \mathbf{F}r4 \wedge \neg(o1 \vee o2 \vee o3 \vee o4 \vee o5))$.

There are four local obstacles labeled uo and three dynamic requests: two *survivor* requests and a $fire$ request. The three dynamic requests have a cyclic motion at a lower speed than that of the robot. The maximum distance traveled by the robot in one discrete time step is $\eta = 1$ (see the steer primitive in Algorithm 7, line 6). The priority function *prior* is defined such that *survivor* request have higher priority than $fire$ request.

A solution to this problem is shown in Figure 3·6. To emphasize working of the on-line planner and simplify the figures for the reader, we chose a simple global transition system $\mathcal{T}_G$. However, the transition system is generated by the off-line algorithm and we present an example of the whole planning framework in the next section. The product automaton has a single accepting state, which corresponds to $x_{accept} = (-7, 0)$ in $\mathcal{T}_G$. The robot must visit x_{accept} infinitely many times and is the

starting and ending point of a surveillance cycle. In each surveillance cycle, the three dynamic requests described above are created. We ran the path planning algorithm in order to complete 100 surveillance cycles. During the simulation, the local path planning algorithm (Algorithm 7) was executed 5947 times. The overall execution time dedicated to local planning (lines 7–8 of Algorithm 6) for a single surveillance cycle was on average 0.743 seconds (std. 0.216, min. 0.436sec, max. 1.645sec). The mean size of the generated local transition system $\mathcal{T}_L$ is 7.6 (std. 13.15, max. 165). The path planning algorithm computed local paths which serviced 292 on-line requests from a total of 296 detected. Thus, we can conclude that the robot was able to satisfy the local mission specification in almost all cases while also ensuring the satisfaction of the global specification.



(a) At the start of the mission

(b) After a few steps

(c) Sensing area with local RRT tree

(d) After completing one surveillance cycle

Figure 3·6: The environment contains four global regions of interest $r1$ (red), $r2$ (green), $r3$ (blue) and $r4$ (magenta), five global obstacles $o1, \ldots, o5$ (dark grey) and four local a priori unknown obstacles labeled uo (light grey). There are also three dynamic requests, two *survivor* (green) and a *fire* (yellow). The circles around the on-line requests delimit their corresponding service area. The sensing range of the robot is shown as a light blue rectangle (length of its side is 5) around the current position of the robot (blue dot), Figure 3·6b. The black arrows and dots represent the global transition system $\mathcal{T}_G$. The trajectory of the robot is shown as a sequence of red arrows. Figure 3·6d shows the trajectory of the robot after completing a surveillance cycle. Figure 3·6c is a close up view of the sensing area of the robot at position $(4.9, 7.3)$ where an RRT tree is generated. The red arrows mark the trajectory of the robot, and the black ones belong to $\mathcal{T}_G$.

Chapter 4

Control in Belief Space with Temporal Logic Specifications

In this chapter, we present a sampling-based algorithm to synthesize control policies with temporal and uncertainty constraints. We introduce a specification language called *Gaussian Distribution Temporal Logic (GDTL)*, an extension of Boolean logic that allows us to incorporate temporal evolution and noise mitigation directly into the task specifications, e.g. "Go to region A and reduce the variance of your state estimate below 0.1 m^2." Our algorithm generates a transition system in the belief space and uses local feedback controllers to break the *curse of history* associated with belief space planning. Furthermore, conventional automata-based methods become tractable. Switching control policies are then computed using a product Markov Decision Process (MDP) between the transition system and the Rabin automaton encoding the task specification. We present algorithms to translate a GDTL formula to a Rabin automaton and to efficiently construct the product MDP by leveraging recent results from incremental computing. Our approach is evaluated in hardware experiments using a camera network and ground robot.

4.1 Gaussian Distribution Temporal Logic

In this section, we define Gaussian Distribution Temporal Logic (GDTL), a predicate temporal logic defined over the space of Gaussian distributions with fixed dimension.

Notation: Let $A \subseteq \mathbb{R}^n$ and $B \subseteq \mathbb{R}^m$, $n, m \geq 0$, we denote by $\mathcal{M}(A, B)$ the set of functions with domain A and co-domain B, where A has positive measure with respect to the Lebesgue measure of $\mathbb{R}^n$. The set of all positive semi-definite matrices of size $n \times n$, $n \geq 1$, is denoted by S^n. $\mathbb{E}[\cdot]$ is the expectation operator. The $m \times n$ zero matrix and the $n \times n$ identity matrix are denoted by $\mathbf{0}_{m,n}$ and $\mathbf{I}_n$, respectively. The supremum and Euclidean norms are denoted by $\|\cdot\|_\infty$ and $\|\cdot\|_2$, respectively.

Let $\mathcal{G}$ denote the Gaussian belief space of dimension n, i.e. the space of Gaussian probability measures over $\mathbb{R}^n$. For brevity, we identify the Gaussian measures with their finite parametrization, mean and covariance matrix. Thus, $\mathcal{G} = \mathbb{R}^n \times S^n$. If $\mathbf{b} = b^0 b^1 \ldots \in \mathcal{G}^\omega$, we denote the suffix sequence $b^i b^{i+1} \ldots$ by $\mathbf{b}^i$, $i \geq 0$.

Definition 4.1 (GDTL Syntax)**.** *The* syntax *of Gaussian Distribution Temporal Logic is defined as*

$$\phi := \top \mid f \leq 0 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathcal{U} \phi_2,$$

where $\top$ is the Boolean constant "True", $f \leq 0$ is a predicate over $\mathcal{G}$, where $f \in \mathcal{M}(\mathcal{G}, \mathbb{R})$, $\neg$ is negation ("Not"), $\wedge$ is conjunction ("And"), and $\mathcal{U}$ is "Until".

For convenience, we define the additional operators: $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\Diamond\, \phi \equiv \top \mathcal{U} \phi$, and $\Box\, \phi \equiv \neg \Diamond \neg\phi$, where $\equiv$ denotes semantic equivalence.

Definition 4.2 (GDTL Semantics)**.** *Let $\mathbf{b} = b^0 b^1 \ldots \in \mathcal{G}^\omega$ be an infinite sequence of belief states. The* semantics *of GDTL is defined recursively as*

$$\begin{aligned}
&\mathbf{b}^i \models \top \\
&\mathbf{b}^i \models f \leq 0 \qquad\qquad \Leftrightarrow \quad f(b^i) \leq 0
\end{aligned}$$

$$
\begin{aligned}
\mathbf{b}^i \models \neg\phi \quad &\Leftrightarrow \quad \neg(\mathbf{b}^i \models \phi) \\
\mathbf{b}^i \models \phi_1 \wedge \phi_2 \quad &\Leftrightarrow \quad (\mathbf{b}^i \models \phi_1) \wedge (\mathbf{b}^i \models \phi_2) \\
\mathbf{b}^i \models \phi_1 \vee \phi_2 \quad &\Leftrightarrow \quad (\mathbf{b}^i \models \phi_1) \vee (\mathbf{b}^i \models \phi_2) \\
\mathbf{b}^i \models \phi_1 \mathcal{U} \phi_2 \quad &\Leftrightarrow \quad \exists j \geq i \ s.t. \ (\mathbf{b}^j \models \phi_2) \wedge (\mathbf{b}^k \models \phi_1, \forall k \in \{i, \ldots j-1\}) \\
\mathbf{b}^i \models \ \Diamond \ \phi \quad &\Leftrightarrow \quad \exists j \geq i \ s.t. \ \mathbf{b}^j \models \phi \\
\mathbf{b}^i \models \ \Box \ \phi \quad &\Leftrightarrow \quad \forall j \geq i \ s.t. \ \mathbf{b}^j \models \phi
\end{aligned}
$$

The word $\mathbf{b}$ *satisfies* ϕ*, denoted* $\mathbf{b} \models \phi$*, if and only if* $\mathbf{b}^0 \models \phi$*.*

By allowing the definition of the atomic predicates used in GDTL to be quite general, we can potentially enforce interesting and relevant properties on the evolution of a system through belief space. Some of these properties include

- Bounds on determinant of covariance matrix $det(P)$. This is used when we want to bound the overall uncertainty about the system's state.
- Bounds on trace of covariance matrix $Tr(P)$. This is used when we want to bound the uncertainty about the system's state in any direction.
- Bounds on state mean $\hat{x}$. This is used when we want to specify where in state space the system should be.

Example 4.1. *Let* R *be a system with Gaussian noise evolving along a straight line with state denoted by* $x \in \mathbb{R}$*. The belief space for this particular robot is thus* $(\hat{x}, P) \in \mathbb{R} \times [0, \infty)$*, where* $\hat{x}$ *and* P *are its state estimate and covariance obtained from its sensors. The system is tasked with going back and forth between two goal regions (denoted as* $\pi_{g,1}$ *and* $\pi_{g,2}$ *in the top of Figure 4·1). It also must ensure that it never overshoots the goal regions or lands in obstacle regions* $\pi_{o,1}$ *and* $\pi_{o,2}$*. The system must*

also maintain a covariance P of less than 0.5 m^2 at all times and less than 0.3 m^2 when in one of the goal regions. These requirements can be described by the GDTL formula

$$
\begin{aligned}
\phi_{1d} \quad &= \phi_{avoid} \wedge \phi_{reach} \wedge \phi_{u,1} \wedge \phi_{u,2} \text{ , where} \\
\phi_{avoid} \quad &= \Box\ \neg((box(\hat{x}, -4, 0.35) \leq 1) \vee (box(\hat{x}, 4, 0.35) \leq 1)) \\
\phi_{reach} \quad &= \Box\ \ \Diamond\ (box(\hat{x}, -2, 0.35) \leq 1) \wedge\ \Box\ \ \Diamond\ (box(\hat{x}, 2, 0.35) \leq 1) \\
\phi_{u,1} \quad &= \Box\ (P < 0.5) \\
\phi_{u,2} \quad &= \Box\ ((box(\hat{x}, -2, 0.35) \leq 1) \wedge (box(\hat{x}, 2, 0.35) \leq 1)) \Rightarrow (P < 0.3)\ ,
\end{aligned}
\tag{4.1}
$$

where $box\,(\hat{x}, x_c, a) = \left\| a^T\,(\hat{x} - x_c) \right\|_\infty$ *is a function bounding* $\hat{x}$ *inside an interval of size* $2\,|a|$ *centered at* x_c*. Subformula* ϕ_{avoid} *encodes keeping the system away from the obstacle regions. Subformula* ϕ_{reach} *encodes periodically visiting the goal regions. Subformula* $\phi_{u,1}$ *encodes maintaining the uncertainty below 0.5* m^2 *globally and subformula* $\phi_{u,2}$ *encodes maintaining the uncertainty below 0.3* m^2 *in the goal regions.*

The belief space associated with this problem is shown in the bottom of Figure 4·1. The curves in the figure correspond to the borders between the satisfaction and violation of predicates in (4.1)*, e.g. the level sets that are induced by the predicates when inequalities are replaced with equality. In the figure, + denotes that the predicate is satisfied in that region and - indicates that it is not. An example belief trajectory that satisfies* (4.1) *is shown in black. Note that every point in this belief trajectory has covariance P less than 0.5, which satisfies* $\phi_{u,1}$*. Further, the forbidden regions in* ϕ_{avoid} *(marked with red stripes) are always avoided while each of the goal regions in* ϕ_{reach} *(marked with green stars) are each visited. Further, whenever the belief is in a goal region, it has covariance P less than 0.3, which means* $\phi_{u,2}$ *is satisfied.*

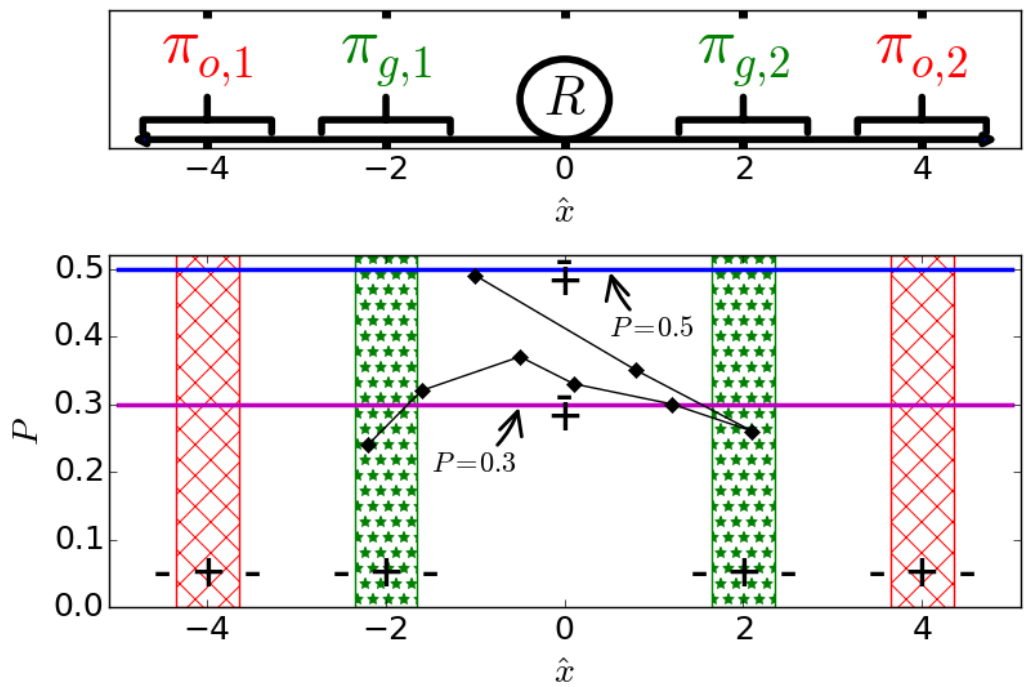


Figure 4·1: (Top) The state space of a system evolving along one dimension and (Bottom) the predicates from (4.1) as functions of the belief of the system from Example 4.1.

4.2 Problem Formulation

In this section, we define the problem of controlling a system to satisfy a given GDTL formula with maximum probability.

4.2.1 Motion and sensing models

We assume the system has noisy linear time invariant (LTI) dynamics given by

$$x_{k+1} = Ax_k + Bu_k + w_k, \tag{4.2}$$

where $x_k \in \mathcal{X}$ is the state of the system, $\mathcal{X} \subseteq \mathbb{R}^n$ is the state space, $A \in \mathbb{R}^{n\times n}$ is the dynamics matrix, $B \in \mathbb{R}^{n\times p}$ is the control matrix, $u_k \in \mathcal{U}$ is a control signal, $\mathcal{U} \subseteq \mathbb{R}^p$ is the control space, and w_k is a zero-mean Gaussian process with covariance

$Q \in \mathbb{R}^{n\times n}$. The state is observed indirectly according to the linear observation model

$$y_k = Cx_k + v_k, \tag{4.3}$$

where $y_k \in \mathcal{Y}$ is a measurement, $\mathcal{Y} \subseteq \mathbb{R}^m$ is the observation space, $C \in \mathbb{R}^{m\times n}$ is the observation matrix and v_k is a zero-mean Gaussian process with covariance $R \in \mathbb{R}^{m\times m}$. We assume the LTI system (4.2), (4.3) is controllable and observable, i.e., (A, B) is a controllable pair and (A, C) is an observable pair. Moreover, we assume that C is full rank. These assumptions apply to many systems, including nonlinear systems that can be linearized to satisfy the assumptions.

The belief state at each time step is characterized by the *a posteriori* state and error covariance estimates, $\hat{x}_k$ and P_k, i.e., $b^k = (\hat{x}_k, P_k)$. The belief state is maintained via a Kalman filter (Kalman, 1960; Bertsekas, 2012), which we denote compactly as

$$b^{k+1} = \tau(b^k, u_k, y_{k+1}), \quad b^0 = (\hat{x}_0, P_0)\ , \tag{4.4}$$

where b^0 is the known initial belief about the system's state centered at $\hat{x}_0$ with covariance P_0. For a belief state $(x, P) \in \mathcal{G}$ we denote the uncertainty ball of radius δ in the belief space centered at (x, P) by $N_\delta(x, P) = \{b \in \mathcal{G} \mid \|b - (x, P)\|_{\mathcal{G}} \leq \delta\}$, where $\|\cdot\|_{\mathcal{G}}$ over $\mathcal{G}$ is a suitable norm in $\mathcal{G}$.

The system model together with the Kalman filter may be represented as a POMDP (Kaelbling et al., 1998; Puterman, 2014; Pineau et al., 2003).

4.2.2 Problem definition

Definition 4.3 (Policy)**.** *A control policy for the system is a feedback function from the belief space $\mathcal{G}$ to the control space, e.g., $\mu : \mathcal{G} \to \mathcal{U}$. Denote the space of all policies by $\mathbb{M} = \mathcal{M}(\mathcal{G},\mathcal{U})$.*

We now introduce the main problem under consideration in this chapter:

Problem 4.1 (Maximum Probability Problem)**.** *Let ϕ be a given GDTL formula and let the system evolve according to dynamics* (4.2)*, with observation dynamics* (4.3)*, and using a Kalman filter defined by* (4.4)*. Find a policy μ^* such that*

$$\begin{aligned} \mu^* &= \arg\max_{\mu \in \mathbb{M}} Pr[\mathbf{b} \models \phi] \\ &\textit{subject to } (4.2), (4.3), (4.4). \end{aligned} \tag{4.5}$$

4.3 Solution

In our approach, we use sampling-based techniques to generate paths throughout the state space. Local controllers drive the systems along these paths and stabilize at key points. The closed-loop behavior of the system induces paths in the belief space. The FIRM describes the stochastic process that generates these paths. We build an MDP by combing the FIRM with a Rabin automaton which then allows us to check if sample paths satisfy a GDTL formula. We compute transition probabilities and intersection probabilities (probability of intersecting a good or bad set from the Rabin automaton's acceptance condition) for each edge in this structure. We use dynamic programming to find the policy in this structure that maximizes the probability of satisfying the formula. The resulting policy can then be translated to a non-stationary switched local controller that approximates the solution to Problem 4.1. An important property of the proposed solution is that all operations are incremental with respect to the size of the FIRM. Note that the proposed solution may be applied to nonlinear systems whose linearizations around random samples in the state space satisfy the assumptions in Section 4.2.1. The details of our solution Algorithm 8 are presented below.

4.3.1 Sampling-based algorithm

We propose a sampling-based algorithm to solve Problem 4.1 that overcomes the curse of dimension and history generally associated with POMDPs. In short, a sampling-based algorithm iteratively grows a graph $\mathcal{T}$ in the state space, where nodes are individual states, and edges correspond to motion primitives that drive the system from state to state (LaValle, 2006). The extension procedure is biased towards exploration of uncovered regions of the state space. Similar to (Agha-mohammadi et al., 2014), we adapt sampling-based methods to produce finite abstractions (e.g., graphs) of the belief space. Algorithm 8 incrementally constructs a transition system $\mathcal{T} = (\mathfrak{B}_{\mathcal{T}}, B_0, \Delta_{\mathcal{T}}, \mathcal{C}_{\mathcal{T}})$, where the state space $\mathfrak{B}_{\mathcal{T}}$ is composed of belief nodes, i.e., bounded hyper-balls in $\mathcal{G}$, $\Delta_{\mathcal{T}}$ is the set of transitions, and $\mathcal{C}_{\mathcal{T}}$ is a set of controllers associated with edges. The center of a belief node is a belief state $b = (x, P^{\infty})$, where the mean x is obtained through random sampling of the system's state space, and P^{∞} is the stationary covariance. The initial belief node is denoted by B_0.

Sampling-based algorithms are built using a set of primitive functions that are assumed to be available:

- $sample(\mathcal{X})$ generates random states from a distribution over the state space $\mathcal{X}$,
- $nearest(x^r, \mathcal{T}) = \arg\min_{x^u}\{\|x^r - x^u\|_2 \mid \exists P^u \wedge N_\delta(x^u, P^u) \in \mathfrak{B}_{\mathcal{T}}\}$ returns the mean x^u of a belief node's center in $\mathcal{T}$ such that x^u is closest to the state x^r using the metric defined on $\mathcal{X}$,
- $near(B_n, \mathfrak{B}_{\mathcal{T}}, \gamma)$ returns the closest γ belief nodes in $\mathfrak{B}_{\mathcal{T}}$ to B_n with respect to the distance between their centers induced by $\|\cdot\|_{\mathcal{G}}$, and
- $steer(x^i, x^t)$ returns a state obtained by attempting to drive the system from x^i towards x^t.

Using these primitive functions, an extension procedure $extend(\mathcal{X}, \mathcal{T})$ of the transition system $\mathcal{T}$ can be defined as:

1. generate a new sample $x^r \leftarrow sample(\mathcal{X})$,
2. find nearest state $x^u \leftarrow nearest(x^r, \mathcal{T})$, and
3. drive the system towards the random sample $x^n \leftarrow steer(x^u, x^r)$.

For more details about sampling-based algorithms, primitive functions and their implementations see (LaValle, 2006; Karaman and Frazzoli, 2011b; Vasile and Belta, 2013) and Chapter 3.

Transitions are enforced using local controllers which are stored in $\mathcal{C}_\mathcal{T}$, i.e., we assign to each edge $e \in \Delta_\mathcal{T}$ a local controller $ec_e \in \mathcal{C}_\mathcal{T}$. Under the assumptions of our model (Agha-mohammadi et al., 2014), the local controllers are guaranteed to stabilize the system to belief nodes along a path in finite time. Thus we abstract the roadmap to a deterministic system. In Algorithm 8, local controllers are generated using the method $localController()$. The design of the node controllers is presented Section 4.4.

The algorithm checks for the presence of a satisfying path using a deterministic Rabin automaton (DRA) $\mathcal{R}$ that is computed from the GDTL specification using an intermediate linear temporal logic (LTL) construction (Jones et al., 2013). There exist efficient algorithms that translate LTL formulae into Rabin automata (Klein and Baier, 2006). We denote the set of predicates in GDTL formula ϕ as F_ϕ.

Definition 4.4 (Rabin Automaton)**.** *A (deterministic) Rabin automaton is a tuple $\mathcal{R} = (S_\mathcal{R}, s_0^\mathcal{R}, \Sigma, \delta, \Omega_\mathcal{R})$, where $S_\mathcal{R}$ is a finite set of states, $s_0^\mathcal{R} \in S_\mathcal{R}$ is the initial state, $\Sigma \subseteq 2^{F_\phi}$ is the input alphabet, $\delta : S_\mathcal{R} \times \Sigma \to S_\mathcal{R}$ is the transition function, and $\Omega_\mathcal{R}$ is a set of tuples $(\mathcal{F}_i, \mathcal{B}_i)$ of disjoint subsets of $S_\mathcal{R}$ which correspond to good ($\mathcal{F}_i$) and bad ($\mathcal{B}_i$) states.*

A transition $s' = \delta(s, \sigma)$ is also denoted by $s \xrightarrow{\sigma}_{\mathcal{R}} s'$. A trajectory of the Rabin automaton $\mathbf{s} = s_0 s_1 \ldots$ is generated by an infinite sequence of symbols $\boldsymbol{\sigma} = \sigma_0 \sigma_1 \ldots$ if $s_0 = s_0^{\mathcal{R}}$ is the initial state of $\mathcal{R}$ and $s_k \xrightarrow{\sigma_k}_{\mathcal{R}} s_{k+1}$ for all $k \geq 0$. Given a state trajectory $\mathbf{s}$ we define $\vartheta_\infty(\mathbf{s}) \subseteq S_{\mathcal{R}}$ as the set of states which appear infinitely many times in $\mathbf{s}$. An infinite input sequence over Σ is said to be accepted by a Rabin automaton $\mathcal{R}$ if there exists a tuple $(\mathcal{F}_i, \mathcal{B}_i) \in \Omega_{\mathcal{R}}$ of good and bad states such that the state trajectory $\mathbf{s}$ of $\mathcal{R}$ generated by $\boldsymbol{\sigma}$ intersects the set $\mathcal{F}_i$ infinitely many times and the set $\mathcal{B}_i$ only finitely many times. Formally, this means that $\vartheta_\infty(\mathbf{s}) \cap \mathcal{F}_i \neq \emptyset$ and $\vartheta_\infty(\mathbf{s}) \cap \mathcal{B}_i = \emptyset$.

4.3.2 Computing transition and intersection probability

Given a transition $e = (B_u, B_v)$ and its associated local controller ec_e, Algorithm 9 computes the transition distribution from an initial DRA state s_u to a some random DRA state, and a set of intersection distributions associated with each pair $(\mathcal{F}_i, \mathcal{B}_i)$ of the acceptance set of $\mathcal{R}$. These distributions are hard to compute analytically. Therefore, we estimate them from sample trajectories of the closed-loop system enforcing edge e. In Algorithm 9, the function $sampleBeliefSet(S)$ returns a random sample from a uniform distribution over the belief set S.

The distribution $\pi^{S_{\mathcal{R}}}$ captures the probability that s_v is the state of $\mathcal{R}$ at the end of closed-loop trajectory generated by controller ec_e to steer the system from belief node B_u and DRA state s_u to belief node B_v: $\pi^{S_{\mathcal{R}}} = Pr[s_v \,|\, e, s_u, ec_e]$, where $s_v \in S_{\mathcal{R}}$, $s_u \overset{\sigma_{0:T-1}}{\rightarrow} s_v$, $b^{0:T} = ec_e(b_u)$, $b_u \in B_u$, and $\sigma_k \leftarrow \{f \,|\, f(b^k) \leq 0, \forall f \in F_\phi\}$.

Each intersection distribution represents the probability that edge e intersects $\mathcal{F}_i$, $\mathcal{B}_i$ or neither, where $(\mathcal{F}_i, \mathcal{B}_i) \in \Omega_{\mathcal{R}}$, and the controller ec_e was used to drive the system along the edge e starting from the DRA state s_u:

Algorithm 8: $ConstructTS(x_0, \phi, \varepsilon)$

Input: initial state x^0, GDTL specification ϕ, and lower bound ε
Output: belief transition system $\mathcal{T}$, product MDP $\mathcal{P}$, and satisfying policy μ^*

```
convert GDTL formula φ to LTL formula ϕ over the set of atomic propositions AP = F_φ
compute DRA R = (S_R, s_0^R, 2^AP, δ, Ω_R) from ϕ
ec_0, P_0^∞ ← localController(x^0)
B_0 ← N_δ(x^0, P_0^∞)
e_0 = (B_0, B_0)
π_0^{S_R}, π_0^{Ω_R} ← computeProb(e_0, s_0, ec_0, R)
initialize belief TS T = (𝔅_T = {B_0}, B_0, Δ_T = {e_0}, C_T = {(e_0, ec_0)})
construct product MDP
P = T × R = (S_P = 𝔅_T × S_R, (B_0, s_0), Act = 𝔅_T, δ_P = {π_0^{S_R}}, Ω_P = {π_0^{Ω_R}})
for index = 1 to N do
    x^n ← extend(X, T)
    ec_n, P_n^∞ ← localController(x^n)
    B_n ← N_δ(x^n, P_n^∞)
    N_n ← near(B_n, 𝔅_T, γ)
    Δ_n ← {(B_i, B_n) | x^n = steer(x^i, x^n), B_i ∈ N_n}
          ∪ {(B_n, B_i) | x^i = steer(x^n, x^i), B_i ∈ N_n}
    𝔅_T ← 𝔅_T ∪ {B_n}, Δ_T ← Δ_T ∪ Δ_n
    S_P ← S_P ∪ ({B_n} × S_R)
    foreach e = (B_u, B_v) ∈ Δ_n do
        C_T ← C_T ∪ {(e, ec_v)}
        foreach s_u ∈ S_R s.t. (B_u, s_u) ∈ S_P do
            π_e^{S_R}, π_e^{Ω_R} ← computeProb(e, s_u, ec_v, R)
            δ_P ← δ_P ∪ {π_e^{S_R}}
            Ω_P ← Ω_P ∪ {π_e^{Ω_R}}
    Δ_P^n = {(p, p') ∈ Δ_P | (p, p')↓_T ∈ Δ_n}
    foreach (F_i, B_i) ∈ Ω_R do // update ECs
        Γ_i = {(p, p') ∈ Δ_P^n | π^{Ω_R}(e, F_i) = 0 ∧ π^{Ω_R}(e, B_i) > 0, e = (p, p')↓_T}
        c_i.update(Δ_P^n \ Γ_i)
    if existsSatPolicy(P) then
        solve DP (4.7) and compute policy μ* with probability of satisfaction p
        if p ≥ ε then return (T, P, μ*)
return (T, P, ∅)
```

Algorithm 9: $computeProb(e = (B_u, B_v), s_u, ec_e, \mathcal{R})$

Input: transition between belief nodes $e = (B_u, B_v)$, starting DRA state s_u, controller enforcing e ec_e, and deterministic Rabin automaton $\mathcal{R}$
Output: transition distribution $\pi^{S_\mathcal{R}}$, and intersection distribution $\pi^{\Omega_\mathcal{R}}$
Parameter: NP – number of particles

```
1  t ← 0_{|S_R|,1}
2  ra_i ← 0_{3,1}, ∀(F_i, B_i) ∈ Ω_R
3  for p = 1 : NP do
4      b_u ← sampleBeliefSet(B_u)
5      b^{0:T} ← ec_e(b_u)
6      for k = 0 to T − 1 do
7          σ_k ← {f | f(b^k) ≤ 0, ∀f ∈ F_φ}
8      s = s_{0:T} ← (s_u --σ_{0:T−1}--> s_T)
9      t[s_T] ← t[s_T] + 1
10     for (F_i, B_i) ∈ |Ω_R| do
11         if F_i ∩ s ≠ ∅ then ra_i[1] ← ra_i[1] + 1
12         if B_i ∩ s ≠ ∅ then ra_i[2] ← ra_i[2] + 1
13         if (F_i ∪ B_i) ∩ s = ∅ then ra_i[3] ← ra_i[3] + 1
14 return (π^{S_R} = t/NP, π^{Ω_R} = {ra_i/NP | 1 ≤ i ≤ |Ω_R|})
```

$$\pi^{\Omega_\mathcal{R}} = \left\{ \left\{ \begin{array}{l} Pr[\mathbf{s} \cap \mathcal{F}_i \mid e, s_u, ec_e] \\ \\ Pr[\mathbf{s} \cap \mathcal{B}_i \mid e, s_u, ec_e] \\ \\ Pr[\mathbf{s} \cap (\mathcal{F}_i \cup \mathcal{B}_i) \mid e, s_u, ec_e] \end{array} \middle| \; \forall (\mathcal{F}_i, \mathcal{B}_i) \in \Omega_\mathcal{R} \right\} \right. \tag{4.6}$$

For convenience, we use the following notation $\pi^{\Omega_\mathcal{R}}(e, X) = Pr[\mathbf{s} \cap X \mid e, s_u, ec_e]$, where $X \in \{\mathcal{F}_i, \mathcal{B}_i, \mathcal{F}_i \cup \mathcal{B}_i\}$.

4.3.3 GDTL-FIRM Product MDP

In this section, we define a construction procedure of the product MDP between the (belief) TS $\mathcal{T}$ and the specification DRA $\mathcal{R}$.

Definition 4.5 (GDTL-FIRM MDP)**.** *Given a DTS* $\mathcal{T} = (\mathfrak{B}_{\mathcal{T}}, B_0, \Delta_{\mathcal{T}}, \mathcal{C}_{\mathcal{T}})$*, a Rabin automaton* $\mathcal{R} = (S_{\mathcal{R}}, s_0^{\mathcal{R}}, \Sigma = 2^{AP}, \delta, \Omega_{\mathcal{R}})$*, and the transition and intersection probabilities* $\pi^{S_{\mathcal{R}}}$*,* $\pi^{\Omega_{\mathcal{R}}}$*, their product MDP, denoted by* $\mathcal{P} = \mathcal{T} \times \mathcal{R}$*, is a tuple* $\mathcal{P} = (S_{\mathcal{P}}, s_0^{\mathcal{P}}, Act, \delta_{\mathcal{P}}, \Omega_{\mathcal{P}})$ *where* $s_0^{\mathcal{P}} = (B_0, s_0^{\mathcal{R}})$ *is the initial state;* $S_{\mathcal{P}} \subseteq \mathfrak{B}_{\mathcal{T}} \times S_{\mathcal{R}}$ *is a finite set of states which are reachable from the initial state by runs of positive probability (see below);* $Act = \mathfrak{B}_{\mathcal{T}}$ *is the set of actions available at each state;* $\delta_{\mathcal{P}} : S_{\mathcal{P}} \times Act \times S_{\mathcal{P}} \to [0,1]$ *is the transition probability defined by* $\delta_{\mathcal{P}}((B_i, s_i), B_j, (B_j, s_j)) = \pi^{S_{\mathcal{R}}}(s_j; e_{ij}, s_i, \mathcal{C}_{\mathcal{T}}(e_{ij}))$*,* $e_{ij} = (B_i, B_j)$*; and* $\Omega_{\mathcal{P}}$ *is the set of tuples of good and bad transitions in the product automaton.*

We denote by $\Delta_{\mathcal{P}} = \big\{\big((B_i, s_i), (B_j, s_j)\big) \,|\, \delta_{\mathcal{P}}((B_i, s_i), B_j, (B_j, s_j)) > 0\big\}$ the set of transitions of positive probability. A transition in $\mathcal{P}$ is also denoted by $p_i \to_{\mathcal{P}} p_j$ if $(p_i, p_j) \in \Delta_{\mathcal{P}}$. A trajectory (or run) of *positive probability* of $\mathcal{P}$ is an infinite sequence $\mathbf{p} = p_0 p_1 \ldots$, where $p_0 = s_0^{\mathcal{P}}$ and $p_k \to_{\mathcal{P}} p_{k+1}$ for all $k \geq 0$.

The acceptance condition for a trajectory of $\mathcal{P}$ is encoded in $\Omega_{\mathcal{P}}$, and is induced by the acceptance condition of $\mathcal{R}$. Formally, $\Omega_{\mathcal{P}}$ is a set of pairs $(\mathcal{F}_i^{\mathcal{P}}, \mathcal{B}_i^{\mathcal{P}})$, where $\mathcal{F}_i^{\mathcal{P}} = \{e \in \Delta_{\mathcal{P}} \,|\, \pi^{\Omega_{\mathcal{R}}}(e, \mathcal{F}_i) > 0\}$, $\mathcal{B}_i^{\mathcal{P}} = \{e \in \Delta_{\mathcal{P}} \,|\, \pi^{\Omega_{\mathcal{R}}}(e, \mathcal{B}_i) > 0\}$, and $(\mathcal{F}_i, \mathcal{B}_i) \in \Omega_{\mathcal{R}}$.

A trajectory of $\mathcal{P} = \mathcal{T} \times \mathcal{R}$ is said to be accepting if and only if there is a tuple $(\mathcal{F}_i^{\mathcal{P}}, \mathcal{B}_i^{\mathcal{P}}) \in \Omega_{\mathcal{P}}$ such that the trajectory intersects the sets $\mathcal{F}_i^{\mathcal{P}}$ and $\mathcal{B}_i^{\mathcal{P}}$ infinitely and finitely many times, respectively. It follows by construction that a trajectory $\mathbf{p} = (B_0, s_0)(B_1, s_1)\ldots$ of $\mathcal{P}$ is accepting if and only if the trajectory $\mathbf{s}^0_{0:T_0-1}\mathbf{s}^1_{0:T_1-1}\cdots$ is accepting in $\mathcal{R}$, where $\mathbf{s}^i_{0:T_i}$ is the random trajectory of $\mathcal{R}$ obtained by traversing the transition $e = (B_i, B_{i+1})$ using the controller $\mathcal{C}_{\mathcal{T}}(e)$ and $s_0^i = s_i$ for all $i \geq 0$. Note that $\mathbf{s}^i_{T_i} = \mathbf{s}^{i+1}_0$. As a result, a trajectory of $\mathcal{T}$ obtained from an accepting trajectory of $\mathcal{P}$ satisfies the given specification encoded by $\mathcal{R}$ with positive probability. We denote the projection of a trajectory $\mathbf{p} = (B_0, s_0)(B_1, s_1)\ldots$ onto $\mathcal{T}$ by $\mathbf{p}\lfloor_{\mathcal{T}} = B_0 B_1 \ldots$. A similar notation is used for projections of finite trajectories.

Remark 4.1. *Note that the product MDP in Definition 4.5 is defined to be amenable to incremental operations with respect to the growth of the DTS, i.e., updating and checking for a solution of positive probability. This property is achieved by requiring the states of $\mathcal{P}$ to be reachable by transitions in $\Delta_{\mathcal{P}}$. The incremental update can be performed using a recursive procedure similar to the one described in (Vasile and Belta, 2013) and Chapter 3.*

Remark 4.2. *The acceptance condition for $\mathcal{P}$ is defined by its transitions and not in the usual way in terms of its states, due to the stochastic nature of transitions between belief nodes in $\mathcal{T}$. We only record the initial and end DRA states of the DRA trajectories induced by the sample paths obtained using the local controllers. Our construction is conservative, but avoids the need to store a (possibly large) number of intermediate states in $\mathcal{P}$ for spurious sample paths deviating from the nominal one.*

4.3.4 Finding satisfying policies

The existence of a satisfying policy with positive probability can be checked efficiently on the product MDP $\mathcal{P}$ by maintaining end components EC[1] for induced subgraphs of $\mathcal{P}$ determined by the pairs in the acceptance condition $\Omega_{\mathcal{P}}$. For each pair $\mathcal{F}_i^{\mathcal{P}}, \mathcal{B}_i^{\mathcal{P}}$, let c_i denote the ECs associated with the graphs $G_i^{\mathcal{P}} = (S_{\mathcal{P}}, \Delta_{\mathcal{P}} \setminus \Gamma_i)$, where $\Gamma_i = \{(p, p') \in \Delta_{\mathcal{P}} \,|\, \pi^{\Omega_{\mathcal{R}}}(e, \mathcal{F}_i) = 0 \wedge \pi^{\Omega_{\mathcal{R}}}(e, \mathcal{B}_i) > 0, e = (p, p')\lfloor_{\mathcal{T}}\}$. Given c_i, checking for a satisfying trajectory in procedure $existsSatPolicy(\mathcal{P})$ becomes trivial. We test if there exists an EC that contains a transition (p, p') such that $\pi^{\Omega_{\mathcal{R}}}(e, \mathcal{F}_i) > 0$, where $e = (p, p')\lfloor_{\mathcal{T}}$. Note that we do not need to maintain $\Omega_{\mathcal{P}}$ explicitly, we only need to maintain the c_i. Efficient incremental algorithms to maintain these ECs were proposed in (Haeupler et al., 2012; Bender et al., 2015)

[1] An EC of an MDP is a sub-MDP such that there exists a policy such that each node in the EC can be reached from each other node in the EC with positive probability.

4.3.5 Dynamic program for Maximum Probability Policy

Given a GDTL-FIRM MDP, we can compute the optimal switching policy to maximize the probability that the given formula ϕ is satisfied. In other words, we find a policy that maximizes the probability of visiting the states in $\mathcal{F}_i$ infinitely often and avoiding $\mathcal{B}_i$. To find this policy, we first decompose $\mathcal{P}$ into a set of end components and find the accepting components. Since any sample path that satisfies ϕ must end in an accepting component, maximizing the probability of satisfying ϕ is equivalent to maximizing the probability of reaching such a component. The optimal policy is thus given by the relationship

$$\begin{aligned} J^{\infty}(s) &= \begin{cases} 1, & s \in c_i \\ \max\limits_{a \in Act(s)} \sum_{s'} \delta(s,a,s') J^{\infty}(s') & \text{else} \end{cases} \\ m(s) &= \arg\max_{a \in Act(s)} \sum_{s'} \delta(s,a,s') J^{\infty}(s') \end{aligned} \tag{4.7}$$

This can be solved by a variety of methods, including approximate value iteration and linear programming (Bertsekas, 2012).

4.3.6 Complexity

The overall complexity of maintaining the ECs used for checking for satisfying runs in $\mathcal{P}$ is $O(|\Omega_{\mathcal{R}}|\,|S_{\mathcal{P}}|^{\frac{3}{2}})$. The complexity bound is obtained using the algorithm described in (Haeupler et al., 2012) and is better by a polynomial factor $|S_{\mathcal{P}}|^{\frac{1}{2}}$ than computing the ECs at each step using a linear algorithm. Thus, checking for the existence of a satisfying run of positive probability can be done in $O(|\Omega_{\mathcal{R}}|)$ time. The dynamic programming algorithm is polynomial in $|S_{\mathcal{P}}|$ (Papadimitriou and Tsitsiklis, 1987).

4.4 Case Studies

In this section, we apply our algorithm to control a unicycle robot moving in a bounded planar environment. To deal with the non-linear nature of the robot model, we locally approximate the robot's dynamics using LTI systems with Gaussian noise around samples in the workspace. This heuristic is very common, since the non-linear and non-Gaussian cases yield recursive filters that do not in general admit finite parametrization. Moreover, the control policy is constrained to satisfy a rich temporal specification. The proposed sampling-based solution overcomes these difficulties due to its randomized and incremental nature. As the size of the GDTL-FIRM increases, we expect the algorithm to return a policy, if one exists, with increasing satisfaction probability. Since it is very difficult to obtain analytical bounds on the satisfaction probability, we demonstrate the performance of our solution in experimental trials.

Motion model

The motion model for our system is a unicycle. We discretize the system dynamics using Euler's approximation. The motion model becomes:

$$x_{k+1} = f(x_k, u_k, w_k) = x_k + \begin{bmatrix} \cos(\theta_k) & 0 \\ \sin(\theta_k) & 0 \\ 0 & 1 \end{bmatrix} \cdot u_k + w_k \tag{4.8}$$

where $x_k = \begin{bmatrix} p_k^x \; p_k^y \; \theta_k \end{bmatrix}^T$, p_k^x, p_k^y and θ_k are the position and orientation of the robot in a global reference frame, $u_k = \begin{bmatrix} v_k' \; \omega_k' \end{bmatrix}^T = \Delta t \begin{bmatrix} v_k \; \omega_k \end{bmatrix}^T$, v_k and ω_k are the linear and rotation velocities of the robot, Δt is the discretization step, and w_k is a zero-mean Gaussian process with covariance matrix $Q \in \mathbb{R}^{3\times 3}$. Next, we linearize the system around a nominal operating point (x^d, u^d) without noise,

$$x_{k+1} = f(x^d, u^d, 0) + A\ (x_k - x^d) + B\ (u_k - u^d) + w_k, \tag{4.9}$$

where $A = \frac{\partial f}{\partial x_k}(x^d, u^d, 0)$ and $B = \frac{\partial f}{\partial u_k}(x^d, u^d, 0)$ are the process and control Jacobians, $x^d = \left[p^{x\,d}\ p^{y\,d}\ \theta^d\right]^T$, and $u^d = \left[v_k'^d\ \omega_k'^d\right]^T$.

In our framework, we associate with each belief node $B_g \in \mathfrak{B}_{\mathcal{T}}$ centered at $(\hat{x}^g, P)$ an LTI system obtained by linearization (4.9) about $(\hat{x}^g, u^g)$, where $u^g = [0.1,\ 0]^T$ corresponds to 0.1 m/s linear velocity and 0 angular velocity.

Observation Model

We localize the robot with a camera network. This reflects the real world constraints of sensor networks, e.g. finite coverage, finite resolution, and improved accuracy with the addition of more sensors. The network was implemented using four TRENDnet Internet Protocol (IP) cameras with known pose with respect to the global coordinate frame of the experimental space. Each 640×400 RGB image is acquired and segmented, yielding multiple pixel locations that correspond to a known pattern on the robot. The estimation of the planar position and orientation of the robot in the global frame is formulated as a least squares problem (*structure from motion*) (Ma et al., 2003). The measurement, $y_k \in \mathcal{Y}$, is given by the discrete observation model: $y_k = Cx_k + v_k$. The measurement error covariance matrix is defined as $R = \mathrm{diag}(r_x, r_y, r_\theta)$, where the value of each scalar is inversely proportional to the number of cameras used in the estimation, i.e. the number of camera views that identify the robot. These values are generated from a camera coverage map (Figure 4·2b) of the experimental space.

Specification

The specification is given over belief states associated with the measurement y of the robot as follows: "Visit regions A and B infinitely many times. If region A is visited, then only corridor D_1 may be used to cross to the right side of the environment.

Similarly, if region B is visited, then only corridor D_2 may be used to cross to the left side of the environment. The obstacle Obs in the center must always be avoided. The uncertainty must always be less than 0.9. When passing through the corridors D_1 and D_2 the uncertainty must be at most 0.6."

The corresponding GDTL formula is:

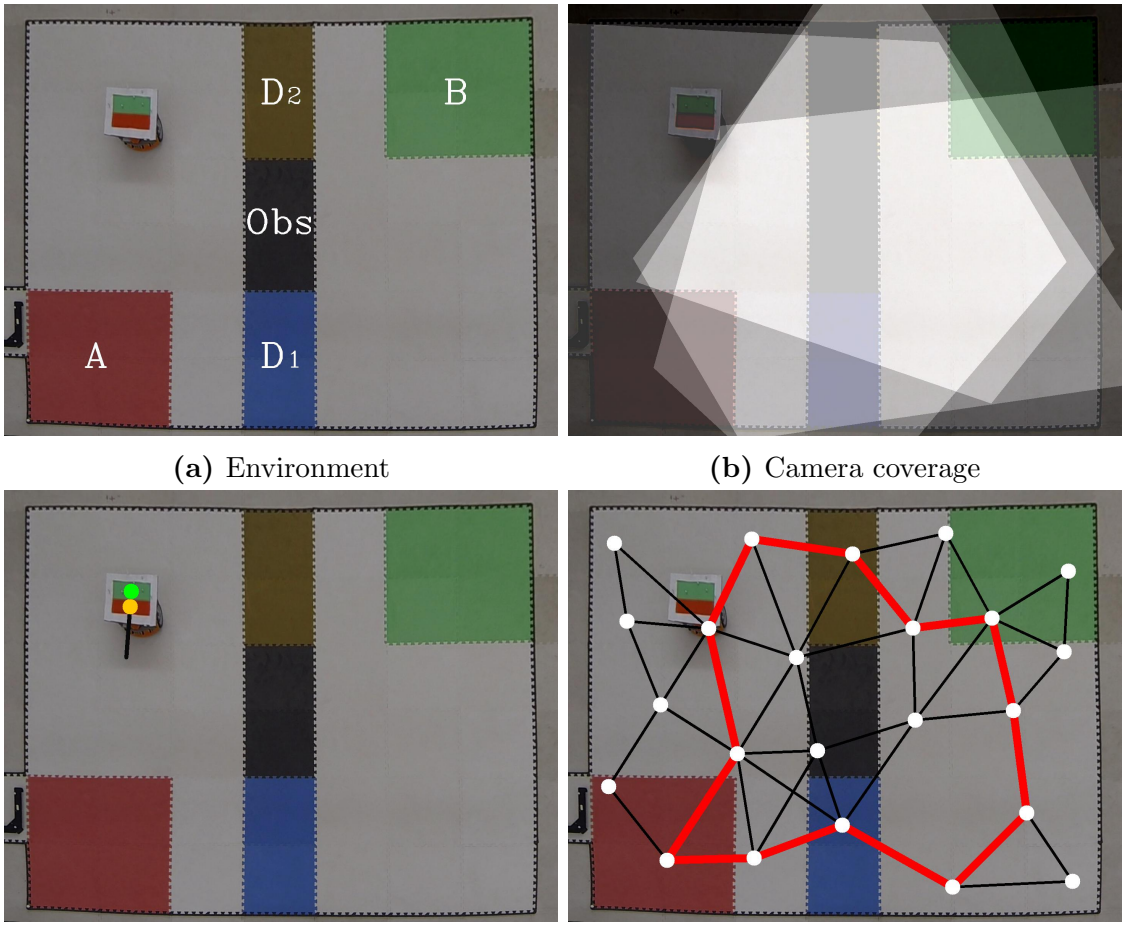
$$\begin{aligned}
\phi_1 &= \phi_{avoid} \wedge \phi_{reach} \wedge \phi_{u,1} \wedge \phi_{u,2} \wedge \phi_{bounds} \qquad (4.10)\\
\phi_{avoid} &= \Box\, \neg\phi_{Obs}\\
\phi_{reach} &= \Box\, \big(\, \Diamond\, (\phi_A \wedge \neg\phi_{D_2} \mathcal{U} \phi_B)\, \Diamond\, (\phi_B \wedge \neg\phi_{D_1} \mathcal{U} \phi_A)\big)\\
\phi_{u,1} &= \Box\, (tr(P) \leq 0.9)\\
\phi_{u,2} &= \Box\, \big((\phi_{D_1} \vee \phi_{D_2}) \Rightarrow (tr(P) \leq 0.6)\big)\\
\phi_{bounds} &= \Box\, (box(\hat{x}, x_c, a) \leq 1),
\end{aligned}$$

where $(\hat{x}, P)$ is a belief state associated with y, $a = \begin{bmatrix} \frac{2}{l} & \frac{2}{w} & 0 \end{bmatrix}$ so that $\hat{x}$ must remain within a rectangular $l \times w$ region with center $x_c = \begin{bmatrix} \frac{l}{2} & \frac{w}{2} & 0 \end{bmatrix}$, $l = 4.13\,m$ and $w = 3.54\,m$. The 5 regions in the environment are defined by GDTL predicate formulae $\phi_{Reg} = (box(\hat{x}, x_{Reg}, r_{Reg}) \leq 1)$, where x_{Reg} and r_{Reg} are the center and the dimensions of region $Reg \in \{A, B, D_1, D_2, Obs\}$, respectively.

Local controllers

We used the following simple switching controller to drive the robot towards belief nodes:

$$u_{k+1} = \begin{cases} \begin{bmatrix} k_D \left\| \alpha^T (x^g - \hat{x}_k) \right\|_2 & k_\theta(\theta_k^{los} - \hat{\theta}_k) \end{bmatrix}^T & \text{if } \left| \theta_k^{los} - \hat{\theta}_k \right| < \frac{\pi}{12} \\ \begin{bmatrix} 0 & k_\theta(\theta_k^{los} - \hat{\theta}_k) \end{bmatrix}^T, & \text{otherwise} \end{cases},$$



(a) Environment **(b)** Camera coverage

(c) Pose estimation **(d)** Transition system

Figure 4·2: Figure a shows an environment with two regions A and B, two corridors D_1 and D_2 and an obstacle Obs. Figure b shows the coverage of the cameras. Figure c shows the pose of the robot computed from the images taken by the 4 cameras. Figure d shows the transition system computed by Algorithm 8.

where $k_D > 0$ and $k_\theta > 0$ are proportional scalar gains, x^g is the goal position, θ_k^{los} is the line-of-sight angle and $\alpha = [1\ 1\ 0]^T$. We assume, as in (Agha-mohammadi et al., 2014), that the controller is able to stabilize the system state and uncertainty around the goal belief state (x^g, P^∞), where P^∞ is the stationary covariance matrix.

Experiments

The algorithms in this paper were implemented in Python2.7 using *LOMAP* (Ulusoy et al., 2013c) and *networkx* (Hagberg et al., 2008) libraries. The *ltl2star* tool (Klein and Baier, 2006) was used to convert the LTL specification into a Rabin automaton. All computation was performed on a Ubuntu 14.04 machine with an Intel Core i7 CPU at 2.4 Ghz and 8GB RAM.

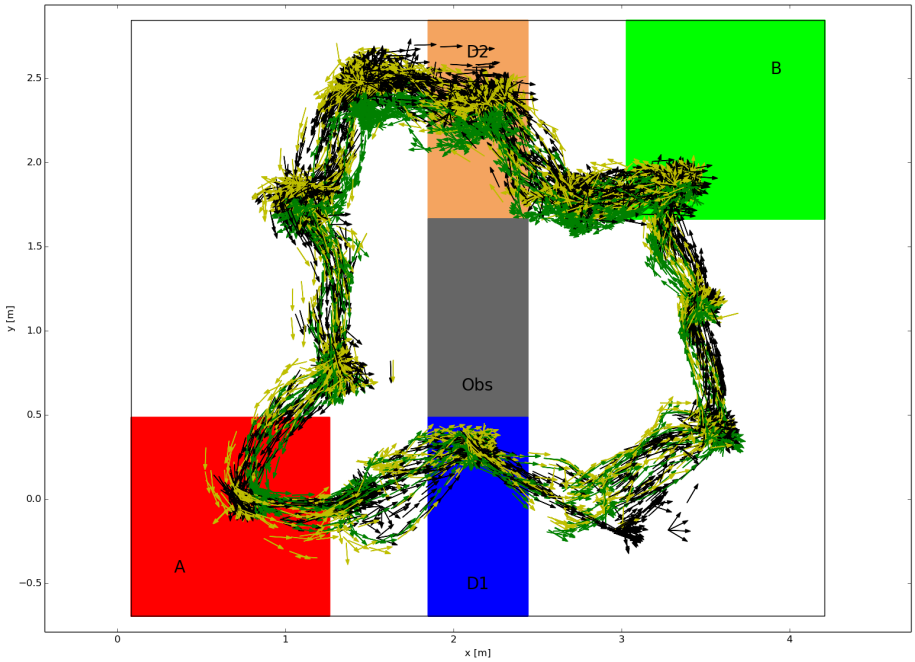


Figure 4·3: The figure shows the trajectory of the robot over 10 surveillance cycles. At each time step, the pose of the robot is marked by an arrow. The true trajectory of the robot is shown in green. The trajectory obtained from the camera network is shown in yellow, while the trajectory estimated by the Kalman filter is shown in black.

A switched feedback policy was computed for the ground robot described by (4.8) operating in the environment shown in Figure 4·2a with mission specification (4.10) using Algorithm 8. The overall computation time to generate the policy was 32.739 seconds and generated a transition system and product MDP of sizes (23, 90) and

(144, 538), respectively. The Rabin automaton obtained from the GDTL formula has 7 states and 23 transitions operating over a set of atomic propositions of size 8. The most computationally intensive operation in Algorithm 8 is the computation of the transition and intersection probabilities. To speed up the execution, we generated trajectories for each transition of the TS and reused them whenever Algorithm 9 is called for a transition of the product MDP. The mean execution time for the probability computation was 0.389 seconds for each transition of $\mathcal{T}$.

We executed the computed policy on the ground vehicle over 9 experimental trials for a total of 24 surveillance cycles. The specification was met in all of surveillance cycles. A trajectory of the ground robot over 10 surveillance cycles (continuous operation) is shown in Figure 4·3.

Chapter 5

Time Window Temporal Logic

This chapter introduces *time window temporal logic* (TWTL), a rich expressivity language for describing various time bounded specifications. In particular, the syntax and semantics of TWTL enable the compact representation of serial tasks, which are prevalent in various applications including robotics, sensor systems, and manufacturing systems. This chapter also discusses the relaxation of TWTL formulae with respect to the deadlines of the tasks. Efficient automata-based frameworks are presented to solve synthesis, verification and learning problems. The key ingredient to the presented solution is an algorithm to translate a TWTL formula to an annotated finite state automaton that encodes all possible temporal relaxations of the given formula. Some case studies are presented to illustrate the expressivity of the logic and the proposed algorithms.

5.1 Preliminaries on Formal Languages

Notation: Given $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^n$, $n \geq 2$, the relationship $\mathbf{x} \sim \mathbf{x}'$, where $\sim \in \{<, \leq, >, \geq\}$, is true if it holds pairwise for all components. $\mathbf{x} \sim a$ denotes $\mathbf{x} \sim a\mathbf{1}_n$, where $a \in \mathbb{R}$ and $\mathbf{1}_n$ is the n-dimensional vector of all ones. The extended set of real numbers is denoted by $\overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$

Definition 5.1 (Prefix language)**.** *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages. We say that $\mathcal{L}_1$ is a prefix language of $\mathcal{L}_2$ if and only if every word in $\mathcal{L}_1$ is a prefix of some*

word in $\mathcal{L}_2$, i.e., for each word $w \in \mathcal{L}_1$ there exists $w' \in \mathcal{L}_2$ such that $w = w'_{0,i}$, where $0 \leq i < |w'|$, The maximal prefix language of a language $\mathcal{L}$ is denoted by $P(\mathcal{L}) = \{w_{0,i} \mid w \in \mathcal{L}, i \in \{0, \ldots, |w| - 1\}\}$.

Definition 5.2 (Unambiguous language)**.** *A language $\mathcal{L}$ is called unambiguous language if no proper subset L of $\mathcal{L}$ is a prefix language of $\mathcal{L} \setminus L$.*

The above definition immediately implies that a word in an unambiguous language can not be the prefix of another word. Moreover, it is easy to show that the converse is also true.

Definition 5.3 (Language concatenation)**.** *Let $\mathcal{L}_1$ be a language over finite words, and let $\mathcal{L}_2$ be a language over finite or infinite words. The concatenation language $\mathcal{L}_1 \cdot \mathcal{L}_2$ is defined as the set of all words ww', where $w \in \mathcal{L}_1$ and $w' \in \mathcal{L}_2$.*

5.2 Time Window Temporal Logic

Time window temporal logic (TWTL) was first introduced in the conference paper (Vasile and Belta, 2014b) as a rich specification language for robotics applications. Besides robotics, TWTL can be used in various domains (e.g., manufacturing, control, software development) that involve specifications with explicit time bounds. In particular, TWTL formulae can express tasks, their durations, and their time windows. TWTL is a linear-time logic encoding sets of discrete-time sequences with values in a finite alphabet.

A TWTL formula is defined over a set of atomic propositions AP and has the following syntax:

$$\phi ::= H^d s \mid H^d \neg s \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi_1 \mid \phi_1 \cdot \phi_2 \mid [\phi_1]^{[a,b]}$$

where s is either the "true" constant $\top$ or an atomic proposition in AP; $\wedge$, $\vee$, and

$\neg$ are the conjunction, disjunction, and negation Boolean operators, respectively; $\cdot$ is the concatenation operator; H^d with $d \in \mathbb{Z}_{\geq 0}$ is the *hold* operator; and $[\ ]^{[a,b]}$ with $0 \leq a \leq b$ is the *within* operator.

The semantics of the operators is defined with respect to the finite subsequences of a (possibly infinite) word $\mathbf{o}$ over 2^{AP}. Let $\mathbf{o}_{t_1,t_2}$ be the subsequence of $\mathbf{o}$, which starts at time $t_1 \geq 0$ and ends at time $t_2 \geq t_1$. The *hold* operator $H^d s$ specifies that $s \in AP$ should be repeated for d time units. The semantics of $H^d \neg s$ is defined similarly, but for d time units only symbols from $AP \setminus \{s\}$ should appear. For convenience, if $d = 0$ we simply write s and $\neg s$ instead of $H^0 s$ and $H^0 \neg s$, respectively. The word $\mathbf{o}_{t_1,t_2}$ satisfies $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, or $\neg\phi$ if $\mathbf{o}_{t_1,t_2}$ satisfies both formulae, at least one formula, or does not satisfy the formula, respectively. The *within* operator $[\phi]^{[a,b]}$ bounds the satisfaction of ϕ to the time window $[a, b]$. The concatenation operator $\phi_1 \cdot \phi_2$ specifies that first ϕ_1 must be satisfied, and then immediately ϕ_2 must be satisfied.

Formally, the semantics of TWTL formulae is defined recursively as follows:

$$
\begin{aligned}
\mathbf{o}_{t_1,t_2} &\models H^d s && \text{iff } s \in o_t, \forall t \in \{t_1, \ldots, t_1 + d\} \wedge (t_2 - t_1 \geq d) \\
\mathbf{o}_{t_1,t_2} &\models H^d \neg s && \text{iff } s \notin o_t, \forall t \in \{t_1, \ldots, t_1 + d\} \wedge (t_2 - t_1 \geq d) \\
\mathbf{o}_{t_1,t_2} &\models \phi_1 \wedge \phi_2 && \text{iff } (\mathbf{o}_{t_1,t_2} \models \phi_1) \wedge (\mathbf{o}_{t_1,t_2} \models \phi_2) \\
\mathbf{o}_{t_1,t_2} &\models \phi_1 \vee \phi_2 && \text{iff } (\mathbf{o}_{t_1,t_2} \models \phi_1) \vee (\mathbf{o}_{t_1,t_2} \models \phi_2) \\
\mathbf{o}_{t_1,t_2} &\models \neg\phi && \text{iff } \neg(\mathbf{o}_{t_1,t_2} \models \phi) \\
\mathbf{o}_{t_1,t_2} &\models \phi_1 \cdot \phi_2 && \text{iff } (\exists t = \arg\min_{t_1 \leq t < t_2} \{\mathbf{o}_{t_1,t} \models \phi_1\}) \wedge (\mathbf{o}_{t+1,t_2} \models \phi_2) \\
\mathbf{o}_{t_1,t_2} &\models [\phi]^{[a,b]} && \text{iff } \exists t \geq t_1 + a \text{ s.t. } \mathbf{o}_{t,t_1+b} \models \phi \wedge (t_2 - t_1 \geq b)
\end{aligned}
$$

A word $\mathbf{o}$ is said to satisfy a formula ϕ if and only if there exists $T \in \{0, \ldots, |\mathbf{o}|\}$ such that $\mathbf{o}_{0,T} \models \phi$.

A TWTL formula ϕ can be verified with respect to a bounded word. Accordingly,

we define the *time bound* of ϕ, i.e., $\|\phi\|$, as the maximum time needed to satisfy ϕ, which can be recursively computed as follows:

$$\|\phi\| = \begin{cases} \max(\|\phi_1\|, \|\phi_2\|) & \text{if } \phi \in \{\phi_1 \wedge \phi_2, \phi_1 \vee \phi_2\} \\ \|\phi_1\| & \text{if } \phi = \neg\phi_1 \\ \|\phi_1\| + \|\phi_2\| + 1 & \text{if } \phi = \phi_1 \cdot \phi_2 \\ d & \text{if } \phi \in \{H^d s, H^d \neg s\} \\ b & \text{if } \phi = [\phi_1]^{[a,b]} \end{cases} \tag{5.1}$$

We denote the language of all words satisfying ϕ by $\mathcal{L}(\phi)$. Note that TWTL formulae are used to specify prefix languages of either Σ^* or Σ^ω, where $\Sigma = 2^{AP}$. Moreover, the number of operators in a TWTL formula ϕ is denoted by $|\phi|$.

Some examples of TWTL formulae for a robot servicing at some regions can be as follows:

- *servicing within a deadline:* "service A for 2 time units before 10",

$$\phi_1 = [H^2 A]^{[0,10]} \text{ and } \|\phi_1\| = 10. \tag{5.2}$$

- *servicing within time windows:* "service A for 4 time units within [3, 8] and B for 2 time units within [4, 7]",

$$\phi_2 = [H^4 A]^{[3,8]} \wedge [H^2 B]^{[4,7]} \text{ and } \|\phi_2\| = 8. \tag{5.3}$$

- *servicing in sequence:* "service A for 3 time units within [0, 5] and after this service B for 2 time units within [4, 9]",

$$\phi_3 = [H^3 A]^{[0,5]} \cdot [H^2 B]^{[4,9]} \text{ and } \|\phi_3\| = 15. \tag{5.4}$$

- *enabling conditions:* "if A is serviced for 2 time units within 9 time units, then B should be serviced for 3 time units within the same time interval (i.e., within 9 time units)",

$$\phi_4 = [H^2 A \Rightarrow [H^3 B]^{[2,5]}]^{[0,9]} \text{ and } \|\phi_4\| = 9, \tag{5.5}$$

where $\Rightarrow$ denotes implication.

In order to describe rich specifications, a temporal logic can be selected based on the expressivity of the logic and the complexity of the corresponding algorithms (e.g., for automata construction). In general, expressivity and complexity are coupled terms such that a logic with very rich expressivity has very high complexity. Furthermore, the easiness to express the specifications and to comprehend the meaning of the formulae is also a crucial aspect when choosing temporal logics. TWTL induces finite languages, and it has the same expressivity of BLTL. On the other hand, STL and MTL are more expressive languages than TWTL since they are developed for real-time systems and can express continuous-time properties.

TWTL provides some benefits over other time-bounded temporal logics. From the perspective of easiness to express specifications and to comprehend formulae, a main benefit of TWTL is the existence of concatenation, within, and hold operators. In particular, these operators lead to compact (shorter length) representation of specifications, which greatly improves the readability of the formulae. For example, consider the specifications in (5.3) and (5.4), which are expressed in various temporal logics in Table 5.1 and 5.2. Note that the TWTL formulae are short and comprehensible whereas an expert in formal methods might be required to create the other formulae to take into account the nested temporal operators, the shifted time windows, and the disjunction of numerous sub-formulae.

From the perspective of complexity, a main benefit of TWTL is the existence of explicit concatenation operator. In particular, the concatenation of two tasks

Table 5.1: The representation of (5.3) in TWTL, BLTL, and MTL.

TWTL	$[H^4 A]^{[3,8]} \wedge [H^2 B]^{[4,7]}$
BLTL	$\mathbf{F}^{\leq 8-4}\mathbf{G}^{\leq 4}A \wedge \mathbf{F}^{\leq 7-2}\mathbf{G}^{\leq 2}B$
MTL	$\bigvee_{i=3}^{8-4} \mathbf{G}_{[i,i+4]}A \wedge \bigvee_{i=4}^{7-2} \mathbf{G}_{[i,i+2]}B$

Table 5.2: The representation of (5.4) in TWTL, BLTL, and MTL.

TWTL	$[H^3 A]^{[0,5]} \cdot [H^2 B]^{[4,9]}$
BLTL	$\mathbf{F}^{\leq 5-3}(\mathbf{G}^{\leq 3}A \wedge \mathbf{F}^{\leq 9-2+3}\mathbf{G}^{\leq 2}B)$
MTL	$\bigvee_{i=0}^{5-3}(\mathbf{G}_{[i,i+3]}A \wedge \bigvee_{j=i+3+4}^{i+3+9-2} \mathbf{G}_{[j,j+2]}B)$

can be expressed in other logics in a more sophisticated way than TWTL. In Table 5.2, we illustrate that the MTL formula contains a set of recursively defined sub-formulae connected by disjunctions whereas the BLTL formula contains nested temporal operators with conjunction. In both cases, dealing with the disjunction of numerous sub-formulae and the nested temporal operators with conjunction significantly increases the complexity of constructing the automaton (i.e., in exponential and quadratic ways, respectively (Maia et al., 2013)). On the other hand, we provide a linear-time algorithm in Section 5.6 to handle the concatenations of tasks under some mild assumptions.

Moreover, the automata construction algorithms in Section 5.6 are specifically developed for TWTL. Thus, an automaton for the satisfying language of a TWTL formula can be constructed directly (without translating it to another logic to use an off-the-shelf tool). For example, the authors of (Tkachev and Abate, 2013) translate a BLTL formula to a syntactically co-safe linear temporal logic (scLTL) formula (Kupferman and Y. Vardi, 2001) to use the automata construction tool *scheck* (Latvala, 2003), which increases the complexity due to additional operations. Finally, for a given TWTL formula ϕ, we show that all possible temporally relaxed ϕ can be encoded to a very compact representation, which is enabled from the definition of

temporal relaxation introduced in the next section.

5.3 Temporal Relaxation

In this section, we introduce a *temporal relaxation* of a TWTL formula. This notion is used in Section 5.4 to formulate an optimization problem over temporal relaxations.

To illustrate the concept of temporal relaxation, consider the following TWTL formula:

$$\phi_1 = [H^1 A]^{[0:2]} \cdot \left[H^3 B \wedge [H^2 C]^{[0:4]}\right]^{[1:8]}. \tag{5.6}$$

In cases where ϕ_1 cannot be satisfied, one question is: what is the "closest" achievable formula that can be performed? Hence, we investigate relaxed versions of ϕ_1. One way to do this is to relax the deadlines for the time windows, which are captured by the *within* operator. Accordingly, a relaxed version of ϕ_1 can be written as

$$\phi_1(\boldsymbol{\tau}) = [H^1 A]^{[0:(2+\tau_1)]} \cdot [H^3 B \wedge [H^2 C]^{[0:(4+\tau_2)]}]^{[1:(8+\tau_3)]}, \tag{5.7}$$

where $\boldsymbol{\tau} = (\tau_1, \tau_2, \tau_3) \in \mathbb{Z}^3$. Note that a critical aspect while relaxing the time windows is to preserve the feasibility of the formula. This means that all sub-formulae of ϕ enclosed by the *within* operators must take less time to satisfy than their corresponding time window durations.

Definition 5.4 (Feasible TWTL formula)**.** *A TWTL formula ϕ is called feasible, if the time window corresponding to each* within *operator is greater than the duration of the corresponding enclosed task (expressed via the* hold *operators).*

Remark 5.1. *Consider the formula in* (5.7)*. For $\phi_1(\boldsymbol{\tau})$ to be a feasible TWTL formula, the following constraint must hold: (i) $2 + \tau_1 \geq 1$; (ii) $4 + \tau_2 \geq 2$ and (iii) $7+\tau_3 \geq \max\{3, 4+\tau_2\}$. Note that $\boldsymbol{\tau}$ may be non-positive. In such cases, $\phi_1(\boldsymbol{\tau})$ becomes a stronger specification than ϕ_1, which implies that the sub-tasks are performed ahead of their actual deadlines.*

Let ϕ be a TWTL formula. Then, a $\tau-$relaxation of ϕ is defined as follows:

Definition 5.5 ($\tau-$Relaxation of ϕ)**.** *Let $\boldsymbol{\tau} \in \mathbb{Z}^m$, where m is the number of* within *operators contained in ϕ. The τ-relaxation of ϕ is a* feasible *TWTL formula $\phi(\boldsymbol{\tau})$, where each subformula of the form $[\phi_i]^{[a_i,b_i]}$ is replaced by $[\phi_i]^{[a_i,b_i+\tau_i]}$.*

Remark 5.2. *For any ϕ, $\phi(\mathbf{0}) = \phi$.*

Definition 5.6 (Temporal Relaxation)**.** *Given ϕ, let $\phi(\boldsymbol{\tau})$ be a feasible relaxed formula. The temporal relaxation of $\phi(\boldsymbol{\tau})$ is defined as $|\boldsymbol{\tau}|_{TR} = \max_j(\tau_j)$.*

Remark 5.3. *If a word $o \models \phi(\boldsymbol{\tau})$ with $|\boldsymbol{\tau}|_{TR} \leq 0$, then $o \models \phi$.*

5.4 Optimization over Temporal Relaxation

In this section, first, we propose a generic optimization problem over temporal relaxations of a TWTL formula. Then, we show how this setup can be used to formulate verification, synthesis, and learning problems.

The objective of the following optimization problem is to find a feasible relaxed version of a TWTL formula that optimizes a cost function penalizing the sets of satisfying and unsatisfying words, and the vector of relaxations.

Problem 5.1. *Let ϕ be a TWTL formula over the set of atomic propositions AP, and let $\mathcal{L}_1$ and $\mathcal{L}_2$ be any two languages over the alphabet $\Sigma = 2^{AP}$. Consider a cost function $F : \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0} \times \mathbb{Z}^m \to \overline{\mathbb{R}}$, where m is the number of* within *operators contained in ϕ. Find $\boldsymbol{\tau}$ such that $F(|\mathcal{L}(\phi(\boldsymbol{\tau})) \cap \mathcal{L}_1|, |\mathcal{L}(\neg\phi(\boldsymbol{\tau})) \cap \mathcal{L}_2|, \boldsymbol{\tau})$ is minimized.*

5.4.1 Verification, synthesis, and learning

In the following, we formulate three specific problems related to verification, synthesis, and learning based on Problem 5.1. The synthesis problem addressed in this

paper follows a recent trend of methods that return policies with reasonable performance even in the case when the specification cannot be met. In literature, some synthesis problems are framed as an optimization problem where the objective is to find a solution satisfying the minimal relaxation of a given specification (Reyes Castro et al., 2013; Tumova et al., 2013a; Tumova et al., 2014; Kim et al., 2015; Guo and Dimarogonas, 2015). Alternatively, some studies impose a hierarchical structure on the input specification based on some given priorities (Reyes Castro et al., 2013; Tumova et al., 2013a; Guo and Dimarogonas, 2015). As such, lower priority properties may be disregarded in case the original specification cannot be satisfied. Yet another approach is presented in (Livingston et al., 2013) where the authors consider a desired global specification and a local specification. Accordingly, the local one defines how the global one may be modified in case of infeasibility. Note that in these approaches it is very hard to translate and evaluate relaxed policies with respect to the original specifications.

The objective of the synthesis problem formulated in this section is to find a control policy (or strategy) that results in the satisfaction of the original formula or its minimal relaxation in case of infeasibility. Our solution approach differs from existing studies (Reyes Castro et al., 2013; Tumova et al., 2013a; Tumova et al., 2014; Kim et al., 2015; Guo and Dimarogonas, 2015) in that the relaxation is defined at a semantic level, i.e., the TWTL formulae are parametrized. The main benefit of our approach is that the results of a synthesis algorithm can be interpreted in the same semantics as the original specification without using an additional representation (e.g., automata) for the relaxed formulae.

The verification problem addressed in this chapter checks if a system satisfies the structure of a specification without considering the time parameters, i.e., the deadlines of the *within* operators. This formulation differs from the generic ones that

consider properties with fixed (temporal or spatial) parameters. Verification problems involving parametric formulae were also considered in (Yang et al., 2012) for STL and in (Alur et al., 2001) for LTL properties. In (Yang et al., 2012), the authors consider a (dense-time) STL specification with a single parameter and the problem of estimating bounds for that parameter. The solution is obtained using an optimization procedure that is defined in terms of robustness degree for STL properties. The problems explored in (Alur et al., 2001) are closer to the ones proposed in this paper. However, both bounded and unbounded properties are considered in (Alur et al., 2001) and the focus of the exposition is geared towards establishing decidability and complexity bounds.

Lastly, we address a parameter learning problem where the goal is to learn the time parameters of a TWTL formula from a given data set. The parameter synthesis for PSTL formulae is tackled in (Asarin et al., 2012; Jin et al., 2015). Moreover, *Temporal Logic Inference*, which is the problem of learning both the structure and parameters of properties, is considered in (Kong et al., 2014; Bombara et al., 2016). In this paper, we focus only on the inference of deadlines for TWTL formulae from labeled data such that the misclassification rate is minimized.

Verification[1]

Given a transition system $\mathcal{T}$ and a TWTL formula ϕ, we want to check if there exists a relaxed formula $\phi(\boldsymbol{\tau})$ such that all output words generated by $\mathcal{T}$ satisfy $\phi(\boldsymbol{\tau})$.

In Problem 5.1, we can set $\mathcal{L}_1 = \emptyset$ and $\mathcal{L}_2 = \mathcal{L}(\mathcal{T})$, and we choose the following cost function:

$$F(x, y, \boldsymbol{\tau}) = 1 - \delta(y), \tag{5.8}$$

[1] This problem is not a verification problem in the usual sense, but rather finding a formula that is satisfied by all runs of a system.

where $x, y \in \mathbb{Z}_{\geq 0}$ and $\delta(x) = \begin{cases} 1 & x = 0 \\ 0 & x \neq 0 \end{cases}$. The cost function in (5.8) has a single global minimum value at 0 which corresponds to the case $\mathcal{L}(\mathcal{T}) \cap \mathcal{L}(\neg\phi(\boldsymbol{\tau})) = \emptyset$.

Synthesis

Given a transition system $\mathcal{T}$ and a TWTL formula ϕ, we want to find a policy (a trajectory of $\mathcal{T}$) that produces an output word satisfying a relaxed version $\phi(\boldsymbol{\tau})$ of the specification with minimal temporal relaxation $|\boldsymbol{\tau}|_{TR}$.

In Problem 5.1, we can set $\mathcal{L}_1 = \mathcal{L}(\mathcal{T})$ and $\mathcal{L}_2 = \emptyset$, and we choose the following cost function:

$$F(x, y, \boldsymbol{\tau}) = \begin{cases} |\boldsymbol{\tau}|_{TR} & x > 0 \\ \infty & \text{otherwise} \end{cases}, \tag{5.9}$$

where $x, y \in \mathbb{Z}_{\geq 0}$. The cost function in (5.9) is minimized by an output word of $\mathcal{T}$, which satisfies the relaxed version of ϕ with minimum temporal relaxation, see Definition 5.6.

Learning

Let ϕ be a TWTL formula and $\mathcal{L}_p$ and $\mathcal{L}_n$ be two finite sets of words labeled as positive and negative examples, respectively. We want to find a relaxed formula $\phi(\boldsymbol{\tau})$ such that the misclassification rate, i.e., $|\{w \in \mathcal{L}_p \mid w \not\models \phi(\boldsymbol{\tau})\}| + |\{w \in \mathcal{L}_n \mid w \models \phi(\boldsymbol{\tau})\}|$, is minimized.

This case can be mapped to the generic formulation by setting $\mathcal{L}_1 = \mathcal{L}_n$, $\mathcal{L}_2 = \mathcal{L}_p$ and choosing the cost function

$$F(x, y, \boldsymbol{\tau}) = x + y, \tag{5.10}$$

which captures the misclassification rate, where $x, y \in \mathbb{Z}_{\geq 0}$.

5.4.2 Overview of the solution

We propose an automata-based approach to solve the verification, synthesis, and learning problems defined above. Specifically, the proposed algorithm constructs an annotated DFA $\mathcal{A}_\infty$, which captures all temporal relaxations of the given formula ϕ, i.e., $\mathcal{L}(\mathcal{A}_\infty) = \mathcal{L}(\phi(\infty))$ (see Definition 5.9 for the definition of $\phi(\infty)$). Note that the algorithm can also be used to construct a (normal) DFA $\mathcal{A}$ which accepts the satisfying language of ϕ, i.e., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\phi)$. Using the resulting DFA $\mathcal{A}_\infty$, we proceed in Section 5.7 to solve the synthesis and verification problems using a product automaton approach. For the synthesis problem, we propose a recursive algorithm that computes a satisfying path with minimum temporal relaxation. The learning problem is solved by inferring the minimum relaxation for each trajectory and then combining these relaxations to ensure minimum misclassification rate.

5.5 Properties of TWTL

In this section, we present properties of TWTL formulae, their temporal relaxations, and their accepted languages.

In this chapter, languages are represented in three ways: as TWTL formulae, as automata, and as sets. As one might expect, there is a duality between some operators of TWTL and set operations, i.e., conjunction, disjunction, and concatenation correspond to intersection, union, and concatenation languages, respectively. Negation may be mapped to complementation with respect to the language of all bounded words, where the bound is given by the time bound of the negated formula.

Proposition 5.4. *The following properties hold*

$$(\phi_1 \cdot \phi_2) \cdot \phi_3 = \phi_1 \cdot (\phi_2 \cdot \phi_3) \tag{5.11}$$

$$\phi_1 \cdot (\phi_2 \vee \phi_3) = (\phi_1 \cdot \phi_2) \vee (\phi_1 \cdot \phi_3) \tag{5.12}$$

$$[\phi_1 \vee \phi_2]^{[a,b]} = [\phi_1]^{[a,b]} \vee [\phi_2]^{[a,b]} \tag{5.13}$$

$$\neg(H^d p) = [\neg p]^{[0,d]} \tag{5.14}$$

$$[\phi_1]^{[a_1,b_1]} = (H^{a_1-1}\top) \cdot [\phi_1]^{[0,b_1-a_1]} \tag{5.15}$$

$$(H^{d_1}p) \cdot (H^{d_2}p) = H^{d_1+d_2+1}p \tag{5.16}$$

$$[\phi_1]^{[a,b]} \Rightarrow [\phi_1]^{[a,b+\tau]} \tag{5.17}$$

$$(\phi_1 \Rightarrow \phi_2) \Rightarrow ([\phi_1]^{[a,b]} \Rightarrow [\phi_2]^{[a,b]}) \tag{5.18}$$

where ϕ_1, ϕ_2, and ϕ_3 are TWTL formulae, $p \in \{s, \neg s\}$, $s \in AP \cup \{\top\}$, and a, b, a_1, b_1, d, d_1, d_2, $\tau \in \mathbb{Z}_{\geq 0}$ such that $a \leq b$ and $1 \leq a_1 \leq b_1$.

Proof. These follow directly from the semantics of TWTL formulae. □

Definition 5.7 (Disjunction-Free Within form)**.** *Let ϕ be a TWTL formula. We say that ϕ is in Disjunction-Free Within (DFW) form if for all* within *operators contained in the formula the associated enclosed subformulae do not contain any disjunction operators.*

An example of a TWTL formula in DFW form is $\phi_1 = [H^2 A]^{[0,9]} \vee [H^5 B]^{[0,9]}$, while a formula not in DFW form is $\phi_2 = [H^2 A \vee H^5 B]^{[0,9]}$. However, ϕ_1 and ϕ_2 are equivalent by (5.13) of Proposition 5.4. The next proposition formalizes this property.

Proposition 5.5. *For any TWTL formula ϕ, if the negation operators are only in front of the atomic propositions, then ϕ can be written in the DFW form.*

Proof. The result follows from the properties of distributivity of Boolean operators and Proposition 5.4, which can be applied iteratively to move all disjunction operators outside the *within* operators. □

In the following, we define the notion of unambiguous concatenation, which enables tracking of progress for sequential specifications. Specifically, if the property holds, then an algorithm is able to decide at each moment if the first specification has finished while monitoring the satisfaction of two sequential specifications.

Definition 5.8. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two languages. We say that the language $\mathcal{L}_1 \cdot \mathcal{L}_2$ is an* unambiguous concatenation *if each word in the resulting language can be split unambiguously, i.e., $\big(L_1, \mathcal{L}_1, \mathcal{L}_1 \cdot (P(\mathcal{L}_2) \setminus \{\epsilon\})\big)$ is a partition of $P(\mathcal{L}_1 \cdot \mathcal{L}_2)$, where $L_1 = \{w_{0,i} \mid w \in \mathcal{L}_1, i \in \{0, \ldots, |w| - 2\}\}$ and $P(L)$ denotes the maximal prefix language of L.*

The three sets of the partition from Definition 5.8 may be thought as indicating whether the first specification is in progress, the first specification has finished, and the second specification is in progress, respectively.

Proposition 5.6. *Consider two languages $\mathcal{L}_1$ and $\mathcal{L}_2$. The language $\mathcal{L}_1 \cdot \mathcal{L}_2$ is an unambiguous concatenation if and only if $\mathcal{L}_1$ is an unambiguous language.*

Proof. Let $\big(L_1, \mathcal{L}_1, \mathcal{L}_1 \cdot (P(\mathcal{L}_2) \setminus \{\epsilon\})\big)$ be a partition of $P(\mathcal{L}_1 \cdot \mathcal{L}_2)$ and L be a proper subset of $\mathcal{L}_1$. Assume that there exists $w \in L$ and $w' \in \mathcal{L}_1 \setminus L$ such that $w = w'_{0,i}$, for some $i \in \{0, \ldots, |w'| - 1\}$. It follows that $w \in L_1$, because $w \neq w'$. However, this contradicts the fact that L_1 and $\mathcal{L}_1$ are disjoint.

Conversely, let $\mathcal{L}_1$ be unambiguous and consider a word $w \in P(\mathcal{L}_1 \cdot \mathcal{L}_2)$. Assume that $w \in L_1 \cap \mathcal{L}_1$. It follows that $\{w\}$ is a prefix language for $\mathcal{L}_1 \setminus \{w\}$, which contradicts with the hypothesis that $\mathcal{L}_1$ is unambiguous. Similarly, if we assume that there exists $w \in P(\mathcal{L}_1) \cap \big(\mathcal{L}_1 \cdot (P(\mathcal{L}_2) \setminus \{\epsilon\})\big)$, then there exists $w', w'' \in \mathcal{L}_1$ such that w' is a prefix of w, w is a prefix of w'', and $|w'| < |w| \leq |w''|$. Thus, we arrive again at a contradiction with the unambiguity of $\mathcal{L}_1$. Thus, the three sets form a partition of $P(\mathcal{L}_1 \cdot \mathcal{L}_2)$. $\square$

In the following results, we frequently use the notion of abstract syntax tree of a TWTL formula. An *Abstract Syntax Tree (AST)* of ϕ is denoted by $AST(\phi)$, where each leaf corresponds to a *hold* operator and each intermediate node corresponds to a Boolean, concatenation, or *within* operator. Given a TWTL formula ϕ, there might exist multiple AST trees that represent ϕ. In this paper, $AST(\phi)$ is assumed to be computed by an LL(*) parser (Parr, 2007). The reader is referred to (Hopcroft et al., 2006) for more details on AST and parsers. An example of an AST tree of (5.6) is illustrated in Figure 5·1.

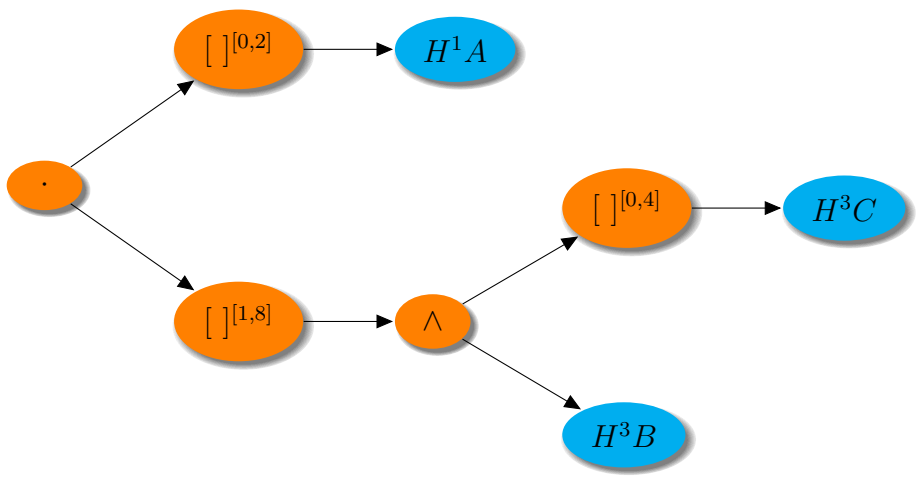


Figure 5·1: An AST corresponding to the TWTL in (5.6). The intermediate orange nodes correspond to the Boolean, concatenation, and *within* operators, while the cyan leaf nodes represent the *hold* operators.

Proposition 5.7. *Let $\boldsymbol{\tau}', \boldsymbol{\tau}'' \in \mathbb{Z}^m$ such that $\phi(\boldsymbol{\tau}')$ and $\phi(\boldsymbol{\tau}'')$ are two feasible relaxed formulae, where m is the number of* within *operators in ϕ. If $\boldsymbol{\tau}' \leq \boldsymbol{\tau}''$, then $\phi(\boldsymbol{\tau}') \Rightarrow \phi(\boldsymbol{\tau}'')$.*

Proof. The proof follows by structural induction over $AST(\phi)$. The base case is trivial, since the leafs correspond to the *hold* operators. For the induction step, the result follows trivially if the intermediate node is associated with a Boolean or concatenation operator. The case of a *within* operator follows from (5.17) and (5.18) in

Proposition 5.4, i.e. $[\phi(\boldsymbol{\tau})]^{[a,b+\tau_1]} \Rightarrow [\phi(\boldsymbol{\tau}')]^{[a,b+\tau_1]} \Rightarrow [\phi(\boldsymbol{\tau}')]^{[a,b+\tau_1']}$, where $a < b \in \mathbb{Z}_{\geq 0}$ and $\boldsymbol{\tau} \leq \boldsymbol{\tau}' \in \mathbb{Z}^m$. We assumed without loss of generality that the first component of the temporal relaxation vectors is assigned to the root node. □

Definition 5.9. *Given an output word* $\mathbf{o}$*, we say that* $\mathbf{o}$ *satisfies* $\phi(\infty)$*, i.e.,* $\mathbf{o} \models \phi(\infty)$*, if and only if* $\exists \boldsymbol{\tau}' < \infty$ *s.t.* $\mathbf{o} \models \phi(\boldsymbol{\tau}')$*.*

The next corollary follows directly from Proposition 5.7.

Corollary 5.8. *Let* $\boldsymbol{\tau} < \infty$*, then* $\phi(\boldsymbol{\tau}) \Rightarrow \phi(\infty)$*,* $\forall \boldsymbol{\tau}$*.*

Proposition 5.9. *Let* $\phi(\boldsymbol{\tau}')$ *and* $\phi(\boldsymbol{\tau}'')$ *be two feasible relaxed formulae. If* $\boldsymbol{\tau}' \leq \boldsymbol{\tau}''$*, then* $\|\phi(\boldsymbol{\tau}')\| \leq \|\phi(\boldsymbol{\tau}'')\|$*.*

Proof. The result follows by structural induction from (5.1) using a similar argument as in the proof of Proposition 5.7. □

An important observation about TWTL is that the accepted languages corresponding to formulae are finite languages. In the following, we characterize such languages in terms of the associated automata.

Definition 5.10. *A DFA is called* strict *if and only if (i) the DFA is blocking, (ii) all states reach a final state, and (ii) all states are reachable from the initial state.*

Proposition 5.10. *Any DFA* $\mathcal{A}$ *may be converted to a strict DFA in* $O(|S_{\mathcal{A}}|)$ *time.*

Proof. States unreachable from the initial state can be identified by traversing the automaton graph from the initial state using either breath- or depth-first search. Similarly, the states not reaching a final state can be removed by traversing the automaton graph using the reverse direction of the transitions. Both operations take at most $O(|\delta_{\mathcal{A}}|) = O(|S_{\mathcal{A}}|)$, since there are at most $|\Sigma|$ transitions outgoing from each state, where Σ is the alphabet of $\mathcal{A}$. □

Note that a strict DFA is not necessarily minimal with respect to the number of states.

Proposition 5.11. *If $\mathcal{L}$ is a finite language over an alphabet Σ, then the corresponding strict DFA is a directed acyclic graph (DAG). Moreover, given a (general) DFA $\mathcal{A}$, checking if its associated language $\mathcal{L}(\mathcal{A})$ is finite takes $O(|S_{\mathcal{A}}|)$ time.*

Proof. For the first part, assume for the sake of contradiction that $\mathcal{A}$ has a cycle. Then, we can form words in the accepted language by traversing the cycle $n \in \mathbb{Z}_{\geq 0}$ times before going to a final state. Note that the states in the cycle are reachable from the initial state and also reach a final state, because $\mathcal{A}$ is a strict DFA. It follows that $\mathcal{L}$ is infinite, which contradicts the hypothesis. Checking if a DFA $\mathcal{A}$ is DAG takes $O(|S_{\mathcal{A}}|)$ by using a topological sorting algorithm, because of the same argument as in Proposition 5.10. □

Corollary 5.12. *Let $\mathcal{L}$ be a finite unambiguous language over the alphabet Σ and $\mathcal{A}$ be its corresponding strict DFA. The following two statements hold:*

1. *if $s \in F_{\mathcal{A}}$, then the set of outgoing transitions of s is empty.*

2. *$\mathcal{A}$ may be converted to a DFA with only one final states.*

Proof. Consider a final state $s \in F_{\mathcal{A}}$. Assume that there exists $s' \in S_{\mathcal{A}}$ such that $s \xrightarrow{\sigma}_{\mathcal{A}} s'$, where $\sigma \in \Sigma$. Since $\mathcal{A}$ is strict, it follows that there is another final state $s'' \in F_{\mathcal{A}}$ which can be reached from s'. Next, we form the words w and w' leading to s and s'' passing trough s', respectively. Clearly, w is a prefix of w', which implies that $\mathcal{L}$ is not an unambiguous language. The second statement follows from the first by noting that in this case, merging all final states does not change the accepted language of the DFA $\mathcal{A}$. □

5.6 Automata construction

In this section, we present a recursive procedure to construct DFAs for TWTL formulae and their temporal relaxations. The resulting DFA are used in Section 5.7 to solve the proposed problems in Section 5.4.1.

Throughout the paper, a TWTL formula is assumed to have the following properties:

Assumption 1. *Let ϕ be a TWTL. Assume that (i) negation operators are only in front of atomic propositions, and (ii) all sub-formulae of ϕ correspond to unambiguous languages.*

The second part (ii) of Assumption 1 is a desired property of specifications in practice, because it is related to the tracking of progress towards the satisfaction of the tasks. More specifically, if (ii) holds, then the end of each sub-formula can be determined unambiguously, i.e., without any look-ahead.

5.6.1 Construction Algorithm

In (Vasile and Belta, 2014b), a TWTL formula ϕ is translated to an equivalent scLTL formula, and then an off-the-shelf tool, such as *scheck* (Latvala, 2003) and *spot* (Duret-Lutz, 2013), is used to obtain the corresponding DFA. In this section, we propose an alternative construction, shown in Algorithm 10, with two main advantages: (i) the proposed algorithm is optimized for TWTL formulae so it is significantly faster than the method used in (Vasile and Belta, 2014b), and (ii) the same algorithm can be used to construct a special DFA, which captures all τ-relaxations of ϕ, i.e., the DFA $\mathcal{A}_\infty$ corresponding to $\phi(\infty)$.

Algorithm 10 constructs the DFA recursively by traversing $AST(\phi)$ computed via an LL(*) parser (Hopcroft et al., 2006; Parr, 2007) from the leaves to the root. If the

Algorithm 10: Translation algorithm – $translate(\cdot)$

Input: ϕ – the specification as a TWTL formula in DFW form
Output: $\mathcal{A}$ – translated DFA

```
1  if φ = φ1 ⊗ φ2, where ⊗ ∈ {∧, ∨, ·} then
2  |  A1 ← translate(φ1), A2 ← translate(φ2)
3  |  A ← ϱ⊗(A1, A2)
4  else if φ = H^d p, where p ∈ {s, ¬s} and s ∈ AP then
5  |  A ← ϱH(p, d, AP)
6  else if φ = [φ1]^[a,b] then
7  |  A1 ← translate(φ1)
8  |  if inf then A ← ϱ∞(A1, a, b)
9  |  else A ← ϱ[](A1, a, b)
10 return A
```

parameter *inf* is true, then the returned DFA is an annotated DFA $\mathcal{A}_\infty$ corresponding to $\phi(\infty)$; otherwise a normal DFA $\mathcal{A}$ is returned. Each operator has an associated algorithm $\varrho_\otimes$ with $\otimes \in \{\wedge, \vee, \cdot, H, \infty, [\,]\}$, which takes the DFAs corresponding to the operands (subtrees of the operator node in the AST) as input. Then, $\varrho_\otimes$ returns the DFA that accepts the formula associated with the operator node. In the following, we present elaborate on all operators and related operations, such as annotating a DFA, relabeling the states of a DFA, or returning the truncated version of a DFA with respect to some given bound.

5.6.2 Annotation

The algorithms presented in this section use DFAs with some additional annotation. In this subsection, we introduce an annotated DFA and two algorithms, Algorithm 12 and Algorithm 11, that are used to (re)label DFAs and the associated annotation data, respectively.

We assume the following conventions to simplify the notation: (i) there is a global boolean variable *inf* accessible by all algorithms, which specifies whether the normal

or the annotated DFAs are to be computed; (ii) in all algorithms, we have $\Sigma = 2^{AP}$; (iii) an element of $\sigma \in \Sigma$ is called a *symbol* and is also a set of atomic propositions, $\sigma \subseteq AP$; (iv) a symbol σ is called *blocking* for a state s if there is no outgoing transition from s activated by σ.

Annotation

An *annotated DFA* is a tuple $\mathcal{A} = (S_{\mathcal{A}}, s_0, \Sigma, \delta, F_{\mathcal{A}}, T_{\mathcal{A}})$, where the first five components have the same meaning as in Definition 2.3 and $T_{\mathcal{A}}$ is a tree that corresponds to the AST of the formula associated with the DFA. Each node T of the tree contains the following information:

1. $T.op$ is the operation corresponding to T;
2. $T.I$ is the set of initial states of the automaton corresponding to T;
3. $T.F$ is the set of final states of the automaton corresponding to T;
4. $T.left$ and $T.right$ are the left and right child nodes of T, respectively.

Additionally, if $T.op$ is $\vee$ (disjunction), then T has another attribute $T.choice$, which is explained in Section Conjunction and disjunction.

Note that the associated trees are set to $\emptyset$ and are ignored, if the normal DFAs are computed, i.e., *inf* is false.

The labels of the states change during the construction of the automata. Algorithm 11 is used to update the labels stored in the data structures of the tree. The algorithm takes the tree T as input, a mapping m from the states to the new labels, and a boolean value e that specifies if the states are mapped to multiple new states. The first step is to convert the states' new labels to singleton sets if e is false (line 1). Then, the algorithm proceeds to process the tree recursively starting with T. The

mapping m is then used to compute $t.I$ and $t.F$ by expanding each state to a set and then computing the union of the corresponding sets (lines 5-6). In the case of $op = \vee$, the three sets B, L, and R, which form the tuple $t.choices$ are also processed. The elements of all three sets are pairs of a state s and a symbol $\sigma \in \Sigma$. Algorithm 11 converts the states of all these pairs in the tree sets (lines 7-12).

Algorithm 11: $relabelTree(T, m, e)$

Input: T – a tree structure
Input: m – (complete) relabeling mapping
Input: e – boolean, true if m maps states to sets of states

1 **if** $\neg e$ **then** $m(s) \leftarrow \{m(s)\}, \forall s$
2 $stack \leftarrow [T]$
3 **while** $stack \neq [\,]$ **do**
4 $\quad t \leftarrow stack.pop()$
5 $\quad t.I \leftarrow \bigcup_{s \in t.I} m(s)$
6 $\quad t.F \leftarrow \bigcup_{s \in t.F} m(s)$
7 $\quad$ **if** $op = \vee$ **then**
8 $\quad\quad B, L, R \leftarrow t.choices$
9 $\quad\quad B' \leftarrow \bigcup_{(s_B,\sigma) \in B} \{(s, \sigma) \mid s \in m(s_B)\}$
10 $\quad\quad L' \leftarrow \bigcup_{(s_L,\sigma) \in L} \{(s, \sigma) \mid s \in m(s_L)\}$
11 $\quad\quad R' \leftarrow \bigcup_{(s_R,\sigma) \in R} \{(s, \sigma) \mid s \in m(s_R)\}$
12 $\quad\quad t.choices \leftarrow (B', L', R')$
13 $\quad$ **if** $t.left \neq \emptyset$ **then** $stack.push(t.left)$
14 $\quad$ **if** $t.right \neq \emptyset$ **then** $stack.push(t.right)$

Relabeling a DFA

The Algorithm 12 relabels the states of a DFA $\mathcal{A}$ with labels given by the mapping m. The map m can be a partial function of the states. The states not specified are labeled with integers starting from i_0 in ascending order. If m is empty, then all states are relabeled with integers. Lastly, if *inf* is true then the tree $T_{\mathcal{A}}$ associated with the DFA is also relabeled, otherwise it is set as empty.

Algorithm 12: $relabel(\mathcal{A}, m, i_0)$

Input: $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta, F_\mathcal{A})$ – a DFA
Input: m – (partial) relabeling mapping
Input: i_0 – start labeling index
Output: the relabeled DFA

1 **for** $s \in S_\mathcal{A}$ *s.t.* $\nexists m(s)$ **do**
2 $\quad m(s) \leftarrow i_0$
3 $\quad i_0 \leftarrow i_0 + 1$
4 $S'_\mathcal{A} \leftarrow \{m(s) \mid s \in S_\mathcal{A}\}$
5 $\delta' \leftarrow \{m(s) \xrightarrow{\sigma}_\mathcal{A} m(s') \mid s \xrightarrow{\sigma}_\mathcal{A} s'\}$
6 $F'_\mathcal{A} \leftarrow \{m(s) \mid s \in F_\mathcal{A}\}$
7 **if** *inf* **then** $T'_\mathcal{A} \leftarrow relabelTree(T_\mathcal{A}, m)$
8 **else** $T'_\mathcal{A} \leftarrow \emptyset$
9 **return** $(S'_\mathcal{A}, m(s_0), \Sigma, \delta', F'_\mathcal{A}, T'_\mathcal{A})$

5.6.3 Operators

Hold

The DFA corresponding to a *hold* operator is constructed by Algorithm 13. The algorithm takes as input an atomic proposition s in positive or negative form, a duration d, and the set of atomic propositions AP. The computed DFA has $d + 2$ states (line 1) that are connected in series as follows: (i) if s is in positive form then the states are connected by all transitions activated by symbols which contain s (lines 2-4); and (ii) if s is in negative form then the states are connected by all transitions activated by symbols which do not contain s (lines 5-7). Lastly, if *inf* is true, a new leaf node is created (line 8).

Conjunction and disjunction

The construction for conjunction and disjunction operations is based on the synchronous product construction and is similar to the standard one (Hopcroft et al., 2006). However, $\varrho_\wedge$ and $\varrho_\vee$ produce strict DFAs, which only have one accepting state.

Algorithm 13: $\varrho_H(p, d, AP)$

Input: $p \in \{s, \neg s\}$, $s \in AP$
Input: d – hold duration
Input: AP – set of atomic propositions
Output: DFA corresponding to $H^d p$

```
1  S ← {0, ..., d+1}
2  if p = s then
3      Σ_s ← 2^AP \ 2^(AP\{s})
4      δ ← {i -σ->_A (i+1) | i ∈ {0, ..., d}, σ ∈ Σ_s}
5  else
6      Σ_¬s ← 2^(AP\{s})
7      δ ← {i -σ->_A (i+1) | i ∈ {0, ..., d}, σ ∈ Σ_¬s}
8  if inf then T ← tree(H^d, ∅, ∅, {0}, {d+1})
9  else T ← ∅
10 return (S, 0, 2^AP, δ, {d+1}, T)
```

Both algorithms recursively construct the product automaton starting from the initial composite state. In the following, we describe the details of the algorithms separately.

Conjunction: The DFA corresponding to the conjunction operation is constructed by Algorithm 14. The procedure is recursive and the synchronization condition, i.e., the transition relation, is the following: given two composite states (s_1, s_2) and (s_1', s_2'), there exists a transition from the first state to the second state if there exists a symbol σ such that: (i) there exists pairwise transitions enabled by σ in the two automata (lines 9-11), i.e., $s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s_1'$ and $s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s_2'$; (ii) one automaton reached a final state and the other has a transition enabled by σ (lines 5-8), i.e., either (a) $s_1 = s_1' = s_{f1}$ and $s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s_2'$, or (b) $s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s_1'$ and $s_2 = s_2' = s_{f2}$. The first case covers the synchronous execution (simulation) of both $\mathcal{A}_1$ and $\mathcal{A}_2$ when a symbol is encountered. The second case corresponds to the situation when the two automata require words of different sizes to accept an input. A simple example of this case is the DFA encoding $H^2 A \wedge H^3 B$ and the input word $\{A, B\}, \{A, B\}, \{A, B\}, \{B\}$, which clearly satisfies the TWTL formula.

Algorithm 14: $\varrho_\wedge(\mathcal{A}_1, \mathcal{A}_2)$

Input: $\mathcal{A}_1 = (S_{\mathcal{A}_1}, s_{01}, \Sigma, \delta_1, \{s_{f1}\}, T_{\mathcal{A}_1})$ – left DFA
Input: $\mathcal{A}_2 = (S_{\mathcal{A}_2}, s_{02}, \Sigma, \delta_2, \{s_{f2}\}, T_{\mathcal{A}_2})$ – right DFA
Output: DFA corresponding to $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$

1 $S \leftarrow \{(s_{01}, s_{02})\}$, $E \leftarrow \emptyset$
2 $stack \leftarrow [(s_{01}, s_{02})]$
3 **while** $stack \neq [\,]$ **do**
4 $\quad \boldsymbol{s} = (s_1, s_2) \leftarrow stack.pop()$
5 $\quad$ **if** $s_1 = s_{f1}$ **then**
6 $\quad\quad S_n \leftarrow \{((s_1, s_2'), \sigma) \mid s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s_2'\}$
7 $\quad$ **else if** $s_2 = s_{f2}$ **then**
8 $\quad\quad S_n \leftarrow \{((s_1', s_2), \sigma) \mid s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s_1'\}$
9 $\quad$ **else**
10 $\quad\quad S_n \leftarrow \{((s_1', s_2'), \sigma) \mid \exists \sigma \in \Sigma$ s.t.
11 $\quad\quad\quad (s_1 \xrightarrow{\sigma}_{\mathcal{A}_1} s_1') \wedge (s_2 \xrightarrow{\sigma}_{\mathcal{A}_2} s_2')\}$
12 $\quad E \leftarrow E \cup \{(\boldsymbol{s}, \sigma, \boldsymbol{s'}) \mid (\boldsymbol{s'}, \sigma) \in S_n\}$
13 $\quad S' \leftarrow \{\boldsymbol{s'} \mid \exists \sigma \in \Sigma$ s.t. $(\boldsymbol{s'}, \sigma) \in S_n\}$
14 $\quad stack.extends(S' \setminus S)$
15 $\quad S \leftarrow S \cup S'$
16 $m_L = \big\{(u, \{(u, v) \in S_{\mathcal{A}}\}) \mid u \in S_{\mathcal{A}_1}\big\}$
17 $m_R = \big\{(v, \{(u, v) \in S_{\mathcal{A}}\}) \mid v \in S_{\mathcal{A}_2}\big\}$
18 $T_{\mathcal{A}} \leftarrow tree(\wedge, relabelTree(T_{\mathcal{A}_1}, m_L, \top),$
19 $\quad relabelTree(T_{\mathcal{A}_2}, m_R, \top), \{(s_{01}, s_{02})\}, \{(s_{f1}, s_{f2})\})$
20 $\mathcal{A} \leftarrow (S, (s_{01}, s_{02}), \Sigma, E, \{(s_{f1}, s_{f2})\}, T_{\mathcal{A}})$
21 **return** $relabel(\mathcal{A}, \emptyset, 0)$

Note that Algorithm 14 generates only composite states which are reachable from the initial composite state (s_{01}, s_{02}). The resulting automaton has a single final state (s_{f1}, s_{f2}) which captures the fact that both automata must accept the input word in order for the product automaton to accept it.

After the automaton is constructed, the corresponding tree is created (lines 16-19). The child subtrees are taken from $\mathcal{A}_1$ and $\mathcal{A}_2$, and relabeled. The relabeling mapping expands each state s to the set of all composite states, which have s as the first or second component corresponding to whether s is a state of the left or right

automaton, respectively.

Disjunction: The disjunction operations is translated using Algorithm 15. The first step of the algorithm is to add a trap state in each of the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ (line 1). All states of an automaton, except the final state, are connected via blocking symbols to the trap state $\bowtie$ (lines 3-4). The trap state has self-transitions for all symbols. Afterwards, the algorithm creates the synchronous product automaton in the same way as for the conjunction operation (lines 4-13). However, in this case, we do not need to treat composite states that contain a final state of one of the two automata separately. This follows from the semantics of the disjunction operation, which accepts a word as soon as at least one automaton accepts the word.

In the standard construction (Hopcroft et al., 2006), the resulting automaton would have multiple final states, which are computed in line 17. However, because finals states do not have outgoing transitions, we can merge all final states and obtain an automaton with only one final state (lines 17-20). The composite trap state is also removed from the set of states (line 18).

The annotation tree is created similarly to the conjunction case (lines 21-24). However, for the disjunction case, we add additional information on the automaton. This information $T.choices$ is used in latter algorithm to determine if a word has satisfied the left, right, or both sub-formulae corresponding to the disjunction formula. This is done by partitioning the transitions incoming into finals states (line 14-16) and storing this partition in the associated tree node (line 25). Note that only the start state and the symbol of each transition is stored in the partition sets and these are well defined, because the DFAs are deterministic.

Algorithm 15: $\varrho_\vee(\mathcal{A}_1, \mathcal{A}_2)$

Input: $\mathcal{A}_1 = (S_{\mathcal{A}_1}, s_{01}, \Sigma, \delta_1, \{s_{f1}\}, T_{\mathcal{A}_1})$ – left DFA
Input: $\mathcal{A}_2 = (S_{\mathcal{A}_2}, s_{02}, \Sigma, \delta_2, \{s_{f2}\}, T_{\mathcal{A}_2})$ – right DFA
Output: DFA corresponding to $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$

1 $S'_{\mathcal{A}_1} \leftarrow S_{\mathcal{A}_1} \cup \{\bowtie\}$, $S'_{\mathcal{A}_2} \leftarrow S_{\mathcal{A}_2} \cup \{\bowtie\}$
2 $\delta'_1 \leftarrow \delta_1 \cup \{(s, \sigma, \bowtie) \mid s \in S'_{\mathcal{A}_1} \setminus \{s_{f1}\}, \sigma \in \Sigma, \nexists \delta_1(s, \sigma)\}$
3 $\delta'_2 \leftarrow \delta_2 \cup \{(s, \sigma, \bowtie) \mid s \in S'_{\mathcal{A}_2} \setminus \{s_{f2}\}, \sigma \in \Sigma, \nexists \delta_2(s, \sigma)\}$
4 $S \leftarrow \{(s_{01}, s_{02})\}$, $E \leftarrow \emptyset$
5 $stack \leftarrow [(s_{01}, s_{02})]$
6 **while** $stack \neq [\,]$ **do**
7 $\quad \boldsymbol{s} = (s_1, s_2) \leftarrow stack.pop()$
8 $\quad S_n \leftarrow \{((s'_1, s'_2), \sigma) \mid \exists \sigma \in \Sigma$ s.t.
9 $\qquad\qquad (s'_1 = \delta'_1(s_1, \sigma)) \wedge (s'_2 = \delta'_2(s_2, \sigma))\}$
10 $\quad E \leftarrow E \cup \{(\boldsymbol{s}, \sigma, \boldsymbol{s'}) \mid (\boldsymbol{s'}, \sigma) \in S_n\}$
11 $\quad S' \leftarrow \{\boldsymbol{s'} \mid \exists \sigma \in \Sigma$ s.t. $(\boldsymbol{s'}, \sigma) \in S_n\}$
12 $\quad stack.extends(S' \setminus S)$
13 $\quad S \leftarrow S \cup S'$
14 $B \leftarrow \{(\boldsymbol{s}, \sigma) \mid \exists \sigma$ s.t. $(\boldsymbol{s}, \sigma, (s_{f1}, s_{f2})) \in E\}$
15 $L \leftarrow \{(\boldsymbol{s}, \sigma) \mid \exists s_2 \neq s_{f2}, \exists \sigma$ s.t. $(\boldsymbol{s}, \sigma, (s_{f1}, s_2) \in E\}$
16 $R \leftarrow \{(\boldsymbol{s}, \sigma) \mid \exists s_1 \neq s_{f1}, \exists \sigma$ s.t. $(\boldsymbol{s}, \sigma, (s_1, s_{f2}) \in E\}$
17 $F \leftarrow \{(s_1, s_2) \in S \mid (s_1 = s_{f1}) \vee (s_2 = s_{f2})\}$
18 $S \leftarrow S \setminus (F \cup \{(\bowtie, \bowtie)\})$
19 $E \leftarrow E \setminus \{(\boldsymbol{s}, \sigma, \boldsymbol{s'}) \in E \mid \boldsymbol{s'} \in F\}$
20 $E \leftarrow E \cup \{(\boldsymbol{s}, \sigma, (s_{f1}, s_{f2})) \mid (\boldsymbol{s}, \sigma) \in B \cup L \cup R\}$
21 $m_L = \big\{(u, \{(u, v) \in S_{\mathcal{A}}\}) \mid u \in S_{\mathcal{A}_1}\big\}$
22 $m_R = \big\{(v, \{(u, v) \in S_{\mathcal{A}}\}) \mid v \in S_{\mathcal{A}_2}\big\}$
23 $T_{\mathcal{A}} \leftarrow tree(\vee, relabelTree(T_{\mathcal{A}_1}, m_L, \top),$
24 $\quad relabelTree(T_{\mathcal{A}_2}, m_R, \top), \{(s_{01}, s_{02})\}, \{(s_{f1}, s_{f2})\})$
25 $T_{\mathcal{A}}.choices \leftarrow (B, L, R)$
26 $\mathcal{A} \leftarrow (S, (s_{01}, s_{02}), \Sigma, E, \{(s_{f1}, s_{f2})\}, T_{\mathcal{A}})$
27 **return** $relabel(\mathcal{A}, \emptyset, 0)$

Concatenation

The algorithm to compute an automaton accepting the concatenation language of two languages is shown in Algorithm 16. The special structure of the unambiguous languages, see Section 5.5 for details, admits a particularly simple and intuitive con-

struction procedure. The composite automaton is obtained by identifying the final state of left automaton $\mathcal{A}_1$ with the initial state of the right automaton $\mathcal{A}_2$.

Algorithm 16: $\varrho.(\mathcal{A}_1, \mathcal{A}_2)$

Input: $\mathcal{A}_1 = (S_{\mathcal{A}_1}, s_{01}, \Sigma, \delta_1, \{s_{f1}\}, T_{\mathcal{A}_1})$ – left DFA
Input: $\mathcal{A}_2 = (S_{\mathcal{A}_2}, s_{02}, \Sigma, \delta_2, \{s_{f2}\}, T_{\mathcal{A}_2})$ – right DFA
Output: DFA corresponding to $\mathcal{L}(\mathcal{A}_1) \cdot \mathcal{L}(\mathcal{A}_2)$

1 $\mathcal{A}_1 \leftarrow relabel(\mathcal{A}_1, \emptyset, 0)$
2 $\mathcal{A}_2 \leftarrow relabel(\mathcal{A}_2, \{(s_{02}, s_{f1})\}, |S_{\mathcal{A}_1}|)$
3 **if** *inf* **then** $T \leftarrow tree(\cdot, T_{\mathcal{A}_1}, T_{\mathcal{A}_2}, \{s_{01}\}, \{s_{f2}\})$
4 **else** $T \leftarrow \emptyset$
5 **return** $(S_{\mathcal{A}_1} \cup S_{\mathcal{A}_2}, s_{01}, \Sigma, \delta_1 \cup \delta_1, \{s_{f2}\}, T)$

Within

There are two algorithms used to construct a DFA associated with a *within* operator, Algorithm 17 and Algorithm 18 correspond to the relaxed and normal construction (lines 6-9 of Algorithm 10).

Relaxed within: The construction procedure Algorithm 17 is as follows: starting from the DFA corresponding to the enclosed formula, all states are connected via blocking symbols to the initial state (lines 3-4). The last step is to create a number of a states connected in sequence for all symbols, similarly to Algorithm 13, and connecting the a-th state to the initial state also for all symbols (lines 5-8).

Connecting all states to the initial state represents a restart of the automaton in case a blocking symbol was encountered. Thus, the resulting automaton offers infinite retries for a word to satisfy the enclosed formula. The a states added before the initial state represent a delay of length a for the start of the tracking of the satisfaction of the enclosed formula. Note that the procedure and resulting automaton do not depend on the upper bound b.

Normal within: The algorithm for the normal case builds upon Algorithm 17. In

Algorithm 17: $\varrho_\infty(\mathcal{A}, a, b)$

Input: $\mathcal{A} = (S_\mathcal{A}, s_0, \Sigma, \delta, \{s_f\}, T_\mathcal{A})$ – child DFA
Input: a – lower bound of time-window
Input: b – upper bound of time-window
Output: computed DFA

$\mathcal{A} \leftarrow relabel(\mathcal{A}, \emptyset, 0)$
$S \leftarrow \emptyset$, $E \leftarrow \emptyset$
for $s \in S_\mathcal{A} \setminus \{s_f\}$ **do**
  $E \leftarrow E \cup \{(s, \sigma, s_0) \mid \nexists s' = \delta(s, \sigma)\}$
if $a > 0$ **then**
  $S \leftarrow \{|S_\mathcal{A}|, \ldots, |S_\mathcal{A}| + a - 1\}$
  $E \leftarrow E \cup \big\{(i, \sigma, i+1) \mid i \in S \setminus \{|S_\mathcal{A}| + a - 1\}, \sigma \in \Sigma\big\}$
  $E \leftarrow E \cup \{(|S_\mathcal{A}| + a - 1, \sigma, s_0) \mid \sigma \in \Sigma\}$
$T \leftarrow tree([\,]_\infty^{[a,b]}, T_\mathcal{A}, \emptyset, \{|S_\mathcal{A}|\}, \{s_f\})$
return $(S_\mathcal{A} \cup S, |S_\mathcal{A}|, \Sigma, \delta \cup E, \{s_f\}, T)$

this case the construction procedure Algorithm 18 must take into account the upper time bound b. Similarly to the relaxed case, we need to restart the automaton of the when a blocking symbol is encountered. However, there are two major differences: (i) the automaton must track the number of restarts, because there are only a finite number of tries depending on the deadline b, and (ii) the automaton $\mathcal{A}$ may need to be truncated for the last restart retries, i.e., all paths must have a length of at most a given length, in order to ensure that the satisfaction is realized before the upper time limit b.

In Algorithm 18, first the maximum number of restarts p is computed in lines 1-2. Then, p DFAs are created (lines 3-12), which correspond to the relabeled and truncated copies of $\mathcal{A}$, see Algorithm 19, and their union is computed iteratively. The truncation bound is computed as the remaining time units until the limit b is reached. The final state is always labeled with -1 (line 7) and, therefore, the resulting DFA has exactly one final state. Next, the restart transitions are added (lines 13-18).

Algorithm 18: $\varrho_{[]}(\mathcal{A}, a, b)$

Input: $\mathcal{A} = (S_{\mathcal{A}}, s_0, \Sigma, \delta, \{s_f\}, T_{\mathcal{A}})$ – child DFA
Input: a – lower bound of time-window
Input: b – upper bound of time-window
Output: computed DFA

```
1  l ← Dijkstra(A, s_0, s_f)
2  p ← b − a − l + 2
3  I ← [ ]                                  // list
4  n ← 0
5  A_r ← (S_{A_r} = ∅, ∞, Σ, δ_r = ∅, ∅, ∅)
6  for k ∈ {1, ..., p} do
7      m ← {(s_f, −1)}                      // mark final state
8      A_a ← relabel(A, m, n)
9      A_t ← truncate(A_a, b − a + 2 − k)
10     A_r ← (S_{A_r} ∪ S_{A_t}, ∞, Σ, δ_r ∪ δ_t, {−1}, ∅)
11     I ← I + [s_{0t}]
12     n ← n + |S_{A_t}|
13 S_c ← {I[0]}, E ← ∅
14 for s_r ∈ I[1 :] do
15     S_n ← ∅
16     for s ∈ S_c \ {−1} do
17         E ← E ∪ {(s, σ, s_r) | σ ∈ Σ s.t. ∄δ_r(s, σ)}
18     S_c ← S_c ∪ {s_r}
19 S ← ∅
20 if a > 0 then
21     S ← {|S_{A_r}|, ..., |S_{A_r}| + a − 1}
22     E ← E ∪ {(i, σ, i + 1) | i ∈ S \ {|S_{A_r}| + a − 1}, σ ∈ Σ}
23     E ← E ∪ {(|S_{A_r}| + a − 1, σ, s_0) | σ ∈ Σ}
24 return (S_{A_r} ∪ S, I[0], Σ, δ_r ∪ E, {−1}, ∅)
```

Note that the transitions, enabled by blocking symbols, lead to initial states of the proper restart automaton. For example, if a blocking symbol was encountered after two symbols, then the restart transition (if it exists) leads to the initial state of the fourth copy of the automaton. Lastly, a delay of a time units is added before the initial state of the automaton similar to the relaxed case.

Truncate

Algorithm 19 takes as input a DFA $\mathcal{A}$ and a cutoff bound l and returns a version of $\mathcal{A}$ with all paths guaranteed to have length at most l. The algorithm is based on a breath-first search and returns a strict DFA.

Algorithm 19: $truncate(\mathcal{A}, l)$

Input: $\mathcal{A} = (S_{\mathcal{A}}, s_0, \Sigma, \delta, \{s_f\}, T_{\mathcal{A}})$ – a DFA
Input: l – cutoff value
Output: computed DFA

```
1  S ← {s_0}
2  E ← ∅
3  L_n ← {s_0}
4  for i ∈ {1, ..., l} do
5      L_c ← L_n
6      L_n ← ∅
7      for s ∈ L_c do
8          for (s_c, σ_c) ∈ {(s', σ) | ∃σ ∈ Σ s.t. s -σ->_A s'} do
9              E ← E ∪ (s, σ_c, s_c))
10             if s_c ∉ S then
11                 S ← S ∪ {s_c}
12                 L_n ← L_n ∪ {s_c}
13 A_t = (S_A, s_0, Σ, δ \ E, {s_f}, T_A)
14 S_traps = {s ∈ S_A | ∄σ ∈ Σ* s.t. s -σ->_{A_t} s_f}
15 return (S_A \ S_traps, s_0, Σ, δ \ E, {s_f}, T_A)
```

5.6.4 Correctness

The following theorems show that the proposed algorithms for translating TWTL formulae to (normal or annotated) automata are correct.

Theorem 5.13. *If ϕ is a TWTL formula satisfying Assumption 1 and the global parameter inf is true, then Algorithm 10 generates a DFA $\mathcal{A}_\infty$ such that $\mathcal{L}(\mathcal{A}_\infty) = \mathcal{L}(\phi(\infty))$.*

Proof. The proof follows by structural induction on $AST(\phi)$ and the properties of TWTL languages.

Before we proceed with the induction, notice that all construction algorithms associated with the operators of TWTL generate strict DFAs with only one final state without any outgoing transitions.

The base case corresponds to the leaf nodes of $AST(\phi)$ which are associated with *hold* operators, see Figure 5·1, and follows by construction from Algorithm 13.

The induction hypothesis requires that the theorem holds for the DFAs returned by the recursion in Algorithm 10. In the case of the conjunction and disjunction operators, the property follows from the product construction method (Hopcroft et al., 2006). The theorem holds also for the concatenation operator, because: (a) the returned DFAs have one final state without any outgoing transitions, and (b) the languages corresponding to the two operand formulae are unambiguous. Thus, the correctness of the construction described in Algorithm 16 follows immediately from the unambiguity of the concatenation, see Definition 5.8. Lastly, the case of the *within* operator (relaxed form), follow from the Assumption 1. The *within* operator adds transitions to a DFA from each state to the initial state on all undefined symbols. In other words, the operator restarts the execution of a DFA from the initial state. If there are no disjunction operators, then going back to the initial state is the only correct choice. Otherwise, because of alternative paths induced by disjunction, there might be other states from which the DFA might need to go back to in order to correctly restart. □

Theorem 5.14. *If ϕ is a TWTL formula satisfying Assumption 1 and the global parameter inf is false, then Algorithm 10 generates DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\phi)$.*

Proof. The proof is similar to that of Theorem 5.13 and is omitted for brevity. □

5.6.5 Complexity

In this section, we review the complexity of the algorithms presented in the previous section for the construction of DFAs from TWTL formulae. The complexity of basic composition operations for incomplete and acyclic DFAs has been explored in (Maia et al., 2013; Han and Salomaa, 2007; Câmpeanu et al., 2001; Gao et al., 2011; Daciuk, 2003). Our construction algorithms differ from the ones in the literature because we specialized and optimized them to translate TWTL formulae and handle words over power sets of atomic propositions.

The complexity of the relabeling procedures are $O(|T|)$ and $O(|S_{\mathcal{A}}|)$ corresponding to Algorithm 11 and Algorithm 12, respectively. The complexity of the *hold* operator Algorithm 13 is $O(d \cdot 2^{|AP|})$. The construction algorithms for conjunction and disjunction Algorithm 14 and Algorithm 15 have the same complexity $O(|S_{\mathcal{A}_1}| \cdot |S_{\mathcal{A}_2}| \cdot 2^{|AP|})$, because these are based on the product automaton construction. Concatenation has complexity $O(|S_{\mathcal{A}_1}| + |S_{\mathcal{A}_2}|)$ due to the relabeling operations. Lastly, the *within* operation can be performed in $O(a \cdot 2^{|AP|} + |S_{\mathcal{A}}| \cdot 2^{|AP|})$ and $O(a \cdot 2^{|AP|} + b\,|S_{\mathcal{A}}| \cdot 2^{|AP|})$ for the infinity Algorithm 17 and the normal Algorithm 18 construction, respectively, where Algorithm 19 used in the normal construction procedure takes $O(|S_{\mathcal{A}}| \cdot 2^{|AP|})$. The overall translation algorithm Algorithm 10 takes at most $O(2^{|\phi|+|AP|})$.

It is very important to notice that the infinity construction does not depend on the deadline b, which makes the procedure more efficient than the normal construction.

5.7 Verification, Synthesis, and Learning Algorithms

In this section, we will use the following notation. Let T be an annotation tree associated with a DFA. We denote by ϕ_T the TWTL formula corresponding to the tree T. Given a finite sequence $\mathbf{p} = p_0, \ldots, p_n$, we denote the first and the last

elements by $b(\mathbf{p}) = p_0$ and $e(\mathbf{p}) = p_n$, respectively.

Definition 5.11 (Primitive)**.** *Let ϕ be a TWTL formula. We say that ϕ is* primitive *if ϕ does not contain any* within *operators.*

5.7.1 Compute temporal relaxation for a word

The automata construction presented in Section 5.6 can be used to compute the temporal relaxation of words with repsect to TWTL formulae. Let ϕ be a TWTL formula and $\boldsymbol{\sigma}$ be a word. In this section, we show how to infer (synthesize) a set of temporal relaxations $\boldsymbol{\tau}$ of the deadlines in ϕ such that $\boldsymbol{\sigma}$ satisfies $\phi(\tau)$ and $|\boldsymbol{\tau}|_{TR}$ is minimized. Algorithm 20 computes the vector of temporal relaxations corresponding to each *within* operator. First, the annotated DFA $\mathcal{A}_\infty$ is computed together with the associated annotation tree T (line 2). Next, additional annotations are added to the tree T using the $initTreeTR()$ procedure (line 3). Each node corresponding to a *within* operation is assigned three variables $T.ongoing$, $T.done$ and $T.steps$, which track whether the processing of the operator is ongoing, done, and the number of steps to process the operator, respectively. The three variables are initialized to $\perp$, $\perp$, and -1, respectively. Then, Algorithm 20 cycles through the symbols of the input word $\boldsymbol{\sigma}$ and updates the tree using $updateTree()$ via Algorithm 21. Finally, the temporal relaxation vector is returned by the $evalTreeTR()$ procedure via Algorithm 22.

The tree is updated recursively in Algorithm 21. A *within* operator is marked as ongoing, i.e., $T.ongoing = \top$, when the current state is in the set of initial states associated with the operator (line 2). Similarly, when the current state is in the set of final states associated with the operator, the *within* operator is marked as done (lines 3-6), i.e. $T.done = \top$ and $T.ongoing = \perp$. The number of steps $T.steps$ of all ongoing *within* operators is incremented (line 7).

To enforce correct computation of the temporal relaxation with respect to the

Algorithm 20: $tr(\cdot)$ – Compute temporal relaxation

Input: $\boldsymbol{\sigma}$ a word over the alphabet 2^{AP}
Input: ϕ a TWTL formula
Output: τ^* - minimum maximal temporal relaxation
Output: $\boldsymbol{\tau}$ - temporal relaxation vector

```
1  if φ is primitive then return (−∞, [ ])
2  A_∞, T ← translate(φ; inf = ⊤)
3  initTreeTR(T)
4  s_prev ←⊥; s_c ← s_0
5  updateTreeTR(T, s_c, s_prev, ∅, ∅)
6  for σ ∈ σ do
7      if s_c ∈ F_{A_∞} then break
8      s_prev ← s_c
9      s_c ← δ_{A_∞}(s_c, σ)
10     updateTreeTR(T, s_c, s_prev, σ, ∅)
11 return evalTreeTR(T)
```

disjunction operators, Algorithm 21 keeps track of a set of constraints C. The set C is composed of state-symbol pairs, and is used to determine which of the two subformulae of a disjunction are satisfied by the input word (lines 12-17). To achieve this, we use the annotation variables $T.choices$ (see Algorithm 15), which capture both cases. For all other operators, the constraint sets are propagated unchanged (lines 8, 10, 11).

Finally, Algorithm 22 extracts the temporal relaxation from the annotation tree T after all symbols of the input word $\boldsymbol{\sigma}$ were processed. Algorithm 22 also computes the minimum maximum temporal relaxation value, which may be $-\infty$ if ϕ is primitive (line 1). The recursion in Algorithm 22 differs between disjunction and the other operators. One subformula is sufficient to hold to satisfy the formula associated with a disjunction operator. Thus, the optimal temporal relaxation is the minimum or maximum between the two optimal temporal relaxations of the subformulae for disjunction (line 12), and conjunction and concatenation (line 13), respectively. Lines

Algorithm 21: $updateTreeTR(\cdot)$

Input: s_c – current state
Input: s_{prev} – previous state
Input: σ – current symbol in word
Input: C – set of constraints associated with the states

```
if T.op = [ ]^[a,b] then
    if s_c ∈ T.I then  T.ongoing ← ⊤
    if s_c ∈ T.F then
        if (C = ∅) ∨ (σ ⊆ C(s_prev)) then
            T.ongoing ←⊥
            T.done ← ⊤
    if T.ongoing then  T.τ ← T.τ + 1
    updateTreeTR(T.left, s_c, s_prev, σ, C)
else
    if T.op = · then  C_L ← ∅; C_R ← C
    else if T.op = ∧ then  C_L ← C; C_R ← C
    else if T.op = ∨ then
        C_L ← T.choices.L ∪ T.choices.B
        C_R ← T.choices.R ∪ T.choices.B
        if C ≠ ∅ then
            C_L ← C ∩ C_L
            C_R ← C ∩ C_R
    updateTreeTR(T.left, s_c, s_prev, σ, C_L)
    updateTreeTR(T.right, s_c, s_prev, σ, C_R)
```

15-16 of Algorithm 22 cover the cases involving primitive subformulae.

The complexity of Algorithm 20 is $O(2^{|\phi|+|AP|} + |\boldsymbol{\sigma}| \cdot |\phi|)$, where the first term is the complexity of constructing $\mathcal{A}_\infty$ in line 1 and the second term corresponds to the update of the tree for each symbol in $\boldsymbol{\sigma}$ and the final evaluation of the tree.

5.7.2 Control policy synthesis for a finite transition system

Let $\mathcal{T}$ be a finite transition system, and ϕ a specification given as a TWTL formula. The procedure to synthesize an optimal control policy by minimizing the temporal relaxation has three steps:

Algorithm 22: $evalTreeTR(\cdot)$

Input: T – annotated tree
Output: τ^* - minimum maximal temporal relaxation
Output: $\boldsymbol{\tau}$ - temporal relaxation vector

```
if φ_T is primitive then return (−∞, [ ])
else if T.op = [φ]^[a,b] then
    τ*_ch, τ_ch = evalTreeTR(tree.left)
    if T.done = ⊤ then
        return (max{τ*_ch, T.steps − b}, [τ_ch, T.steps − b])
    else
        return (−∞, [τ_ch, −∞])
else // ∧, ∨ or ·
    τ*_L, τ_L = evalTreeTR(tree.left)
    τ*_R, τ_R = evalTreeTR(tree.right)
    if (τ*_L ≠ −∞) ∧ (τ*_R ≠ −∞) then
        if T.op = ∨ then τ* ← min{τ*_L, τ*_R}
        else τ* ← max{τ*_L, τ*_R}
    else
        if T.op = ∨ then τ* ← max{τ*_L, τ*_R}
        else τ* ← −∞
    return (τ*, [τ_L, τ_R])
```

1. constructing the annotated DFA $\mathcal{A}_\infty$ corresponding to ϕ,

2. constructing the synchronous product $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\infty$ between the transition system $\mathcal{T}$ and the annotated DFA $\mathcal{A}_\infty$,

3. computing the optimal policy on $\mathcal{P}$ using Algorithm 23 and generating the optimal trajectory of $\mathcal{T}$ from the optimal trajectory of $\mathcal{P}$ by projection.

Before we present the details of the proposed algorithm, we want to point out that completeness may be decided easily by using the product automaton $\mathcal{P}$. That is, testing if there exists a temporal relaxation such that a satisfying policy in $\mathcal{T}$ may be synthesized can be performed very efficiently as shown by the following theorem.

Theorem 5.15. *Let ϕ be a TWTL formula and $\mathcal{T}$ be a finite transition system. Deciding if there exists a finite $\boldsymbol{\tau} \in \mathbb{Z}^m$ and a trajectory $\mathbf{x}$ of $\mathcal{T}$ such that $\mathbf{o} \models \phi(\boldsymbol{\tau})$, can be performed in $O(|\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$, where m is the number of* within *operators in ϕ, $\mathcal{A}_\infty$ is the annotated DFA corresponding to ϕ, $\mathbf{o}$ is the output trajectory induced by $\mathbf{x}$, and Δ and $\delta_{\mathcal{A}_\infty}$ are the sets of transitions of $\mathcal{T}$ and $\mathcal{A}_\infty$, respectively.*

Remark 5.16. *The complexity in Theorem. 5.15 is independent of the deadlines of the* within *operators ϕ.*

Proof. The result follows immediately using Dijkstra's algorithm on the product automaton $\mathcal{P}$. □

Note that Dijkstra's algorithm may not necessarily provide an optimal trajectory of $\mathcal{T}$ with respect to the minimum maximum temporal relaxation of the induced output word. Thus, we present a Dijkstra-based procedure to compute an optimal policy using the product automaton $\mathcal{P}$. The proposed solution is presented in Algorithm 23, which describes a recursive procedure over an annotated AST tree T.

The recursive procedure in Algorithm 23 has six cases. The first case (lines 1-3) corresponds to a primitive formula. In this case, there are no deadlines to relax since the formula does not contain any *within* operators. Thus, solutions (if any exist) can be computed using Dijkstra's algorithm. The next two cases treat the *within* operators. In the former case (lines 4-5), the enclosed formula is a primitive formula and the only deadline which must be optimized is the one associated with the current *within* operator. In the latter case (lines 7-10), the enclosed formula is not primitive. Therefore, there are multiple deadlines that must be considered. To optimize the temporal relaxation $|\cdot|_{TR}$, we take the maximum between the previous maximum temporal relaxation and the current temporal relaxation (line 10). The fourth case (lines 11-15) handles the concatenation operator. First, the paths and the corre-

Algorithm 23: Policy synthesis – $policy(T, \mathcal{P})$

Input: T – the annotation AST tree
Input: $\mathcal{P}$ – product automaton

1 **if** ϕ_T *is primitive* **then**
2 $\quad M = \{\mathbf{p} \mid b(\mathbf{p}) \in T.I, e(\mathbf{p}) \in T.F\}$
3 $\quad \tau^*[\mathbf{p}] = -\infty, \forall \mathbf{p} \in M$
4 **else if** $T.op = [\,]^{[a,b]} \wedge \phi_{T.left}$ *is primitive* **then**
5 $\quad M = \{\mathbf{p} \mid b(\mathbf{p}) \in T.I, e(\mathbf{p}) \in T.F\}$
6 $\quad \tau^*[\mathbf{p}] = |\mathbf{p}| - b, \forall \mathbf{p} \in M$
7 **else if** $T.op = [\,]^{[a,b]} \wedge \phi_{T.left}$ *is not primitive* **then**
8 $\quad M_{ch}, \tau^{max}_{ch} = policy(T.left, \mathcal{P})$
9 $\quad M = \{p_i \xrightarrow{a} p \xrightarrow{*} p' \mid p_i \in T.I, p \xrightarrow{*} p' \in M_{ch}\}$
10 $\quad \tau^*[\mathbf{p}] = \max\{|\mathbf{p}| - b, \tau^*_{ch}[\mathbf{p}]\}, \forall \mathbf{p} \in M$
11 **else if** $T.op = \cdot$ **then**
12 $\quad M_L, \tau^*_L = policy(T.left, \mathcal{P})$
13 $\quad M_R, \tau^*_R = policy(T.right, \mathcal{P})$
14 $\quad M = \{\mathbf{p}_1 \cdot \mathbf{p}_2 \mid \mathbf{p}_1 \in M_L, \mathbf{p}_2 \in M_R, e(\mathbf{p}_1) \rightarrow_{\mathcal{P}} b(\mathbf{p}_2)\}$
15 $\quad \tau^*[\mathbf{p}] = \max\{\tau^*_L(\mathbf{p}), \tau^*_R(\mathbf{p})\}, \forall \mathbf{p} \in M$
16 **else if** $T.op = \vee$ **then**
17 $\quad M_L, \tau^*_L = policy(T.left, \mathcal{P})$
18 $\quad M_R, \tau^*_R = policy(T.right, \mathcal{P})$
19 $\quad M = M_L \cup M_R$
20 $\quad \tau^*[\mathbf{p}] = \begin{cases} \tau^*_L[\mathbf{p}] & \mathbf{p} \in M \setminus M_R \\ \tau^*_R[\mathbf{p}] & \mathbf{p} \in M \setminus M_L \\ \min\{\tau^*_L[\mathbf{p}], \tau^*_R[\mathbf{p}]\} & \mathbf{p} \in M_L \cap M_R \end{cases}$
21 **else if** $T.op = \wedge$ **then**
22 $\quad M_L, \tau^*_L = policy(T.left, \mathcal{P})$
23 $\quad M_R, \tau^*_R = policy(T.right, \mathcal{P})$
24 $\quad M = M_L \cap M_R$
25 $\quad \tau^*[\mathbf{p}] = \max\{\tau^*_L(\mathbf{p}), \tau^*_R(\mathbf{p})\}, \forall \mathbf{p} \in M$
26 **return** (M, τ^*)

sponding temporal relaxations are computed for the left and the right subformulae in lines 12 and 13, respectively. Afterwards, the paths satisfying the left subformula are concatenated to the paths satisfying the right formula. However, the concatenation

of paths p_L and p_R is restricted to pairs which have the following property: there exists a transition in $\mathcal{P}$ between the last state of p_L and the first state in p_R. The temporal relaxation of the concatenation of two paths is the maximum between the temporal relaxations of the two paths (line 15). The next case is associated with the disjunction operator (lines 16-20). As in the concatenation case, first the paths satisfying the left M_L and the right M_R subformulae are computed in lines 17 and 18, respectively. The set corresponding to the disjunction of the two formulae is the union of the two sets because the paths must satisfy either one of the two subformulae. The temporal relaxation of a path p in the union is computed as follows (line 20): (a) if a path is only in the left, $\mathbf{p} \in M_L \setminus M_R$, or only in the right set, $\mathbf{p} \in M_R \setminus M_L$, then the temporal relaxation is $\tau_L^*[\mathbf{p}]$ or $\tau_R^*[\mathbf{p}]$, respectively; (b) the path is in both sets, $\mathbf{p} \in M_L \cap M_R$, then the temporal relaxations is the minimum of the two previously computed ones, $\min\{\tau_L^*[\mathbf{p}], \tau_R^*[\mathbf{p}]\}$. In the case (a), $\mathbf{p}$ satisfies only one subformula and, therefore, only one temporal relaxation is available. In the case (b), $\mathbf{p}$ satisfies both subformulae. Because only one is needed, the subformula that yields the minimum temporal relaxation is chosen, i.e., the minimum between the two temporal relaxations. The last case handles the conjunction operator (lines 21-25). As in the previous two cases, the paths satisfying the left and the right subformulae are computed first (lines 22-23). Then the intersection of the two sets is computed as the set of paths satisfying the conjunctions because the paths must satisfy both subformulae. The temporal relaxations of the paths in the intersections are computed as the maxima between the previously computed temporal relaxations for the left and the right subformulae.

Note that considering primitive formulae in Algorithm 23, instead of traversing the AST all the way to the leaves, optimizes the running time and the level of recursion of the algorithm.

A very important property of Algorithm 23 is that its complexity does not depend on the deadlines associated with the *within* operators of the TWTL specification formula ϕ. This is an immediate consequence of the DFA construction proposed in Section 5.6. Moreover, it follows from Remark 5.3 that the completeness with respect to ϕ (unrelaxed) may also be decided independently of the values of the deadline values. Formally, we have the following results.

Theorem 5.17. *Let ϕ be a TWTL formula and $\mathcal{T}$ be a finite transition system. Synthesizing a trajectory $\mathbf{x}$ of $\mathcal{T}$ such that $\mathbf{o} \models \phi(\boldsymbol{\tau})$ and $|\boldsymbol{\tau}|_{TR}$ is minimized can be performed in $O(|\phi| \cdot |\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$, where $\boldsymbol{\tau} \in \mathbb{Z}^m$, m is the number of* within *operators in ϕ, $\mathcal{A}_\infty$ is the annotated DFA corresponding to $\phi(\infty)$, $\mathbf{o}$ is the output trajectory induced by $\mathbf{x}$, and Δ and $\delta_{\mathcal{A}_\infty}$ are the sets of transitions of $\mathcal{T}$ and $\mathcal{A}_\infty$, respectively.*

Proof. The worst-case complexity of Algorithm 23 is achieved when the TWTL formula ϕ has the form of primitive formulae enclosed by *within* operators and then composed by either the conjunction, disjunction, and concatenation operators.

The recursive algorithm stops when it encounters the primitive formulae and executes Dijkstra's algorithm that takes at most $O(|\Delta_{\mathcal{P}}|) = O(|\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$ time. Since the recursion is performed with respect to an AST T of ϕ, the algorithm processes each operator only once. The complexity bound follows because the size of the set of paths M returned by the algorithm is at most the sum of the sized of the sets corresponding to the left and the right sets M_L and M_R, respectively. Thus, we obtain the bound $O(|\phi| \cdot |\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$ by summing up the time complexity over all nodes of T. □

Corollary 5.18. *Let ϕ be a TWTL formula and $\mathcal{T}$ be a finite transition system. Deciding if there exists a trajectory $\mathbf{x}$ of $\mathcal{T}$ such that $\mathbf{o} \models \phi$ can be performed in $O(|\phi| \cdot |\Delta| \cdot |\delta_{\mathcal{A}_\infty}|)$, where $\mathcal{A}_\infty$ is the annotated DFA corresponding to ϕ, $\mathbf{o}$ is the*

output trajectory induced by $\mathbf{x}$*, and* Δ *and* $\delta_{\mathcal{A}_\infty}$ *are the sets of transitions of* $\mathcal{T}$ *and* $\mathcal{A}_\infty$*, respectively.*

Proof. It follows from Theorem. 5.17 and Remark 5.3. □

5.7.3 Verification

The procedure described in Algorithm 24 solves the verification problem of a transition system $\mathcal{T}$ against all relaxed versions of a TWTL specification First, the annotated DFA $\mathcal{A}_\infty$ corresponding to ϕ is computed (line 1). Then a trap state $\bowtie$ is added in line 2 (see Algorithm 15 for details). Note that the final state is not connected to trap state. The transition system $\mathcal{T}$ is composed with the DFA $\mathcal{A}_\infty$ to produce the product automaton $\mathcal{P}$ (line 3).

Lastly, it is checked if all trajectories of $\mathcal{P}$ reach the final state in finite time (line 4), i.e., satisfy a relaxation of ϕ. The condition in line 4 ensures that: (i) there are final states; (ii) all paths are finite, i.e., $\mathcal{P}$ is a DAG; and (iii) the only allowed sink states are the final states, i.e., the out degree $\deg(v)$ of all non-final states v of $\mathcal{P}$ is positive.

Algorithm 24: Verification

Input: $\mathcal{T}$ – transition system
Input: ϕ – TWTL specification
Output: Boolean value

1 $\mathcal{A}_\infty \leftarrow translate(\phi; \inf\ = \top)$
2 add trap state $\bowtie$ to $\mathcal{A}_\infty$
3 $\mathcal{P} \leftarrow \mathcal{T} \times \mathcal{A}_\infty$
4 **return** $F_\mathcal{P} \neq \emptyset \wedge isDAG(\mathcal{P}) \wedge \left(\bigwedge_{p \in S_\mathcal{P} \setminus F_\mathcal{P}} \deg(v) > 0\right)$

5.7.4 Learning deadlines from data

In this section, we present a simple heuristic procedure to infer deadlines from a finite set of labeled traces such that the misclassification rate is minimized. Let ϕ be a TWTL formula and $\mathcal{L}_p$ and $\mathcal{L}_n$ be two finite sets of words labeled as positive and negative examples, respectively. The misclassification rate is $|\{w \in L_p \mid w \not\models \phi(\boldsymbol{\tau})\}| + |\{w \in L_n \mid w \models \phi(\boldsymbol{\tau})\}|$, where $\phi(\boldsymbol{\tau})$ is a feasible $\boldsymbol{\tau}$-relaxation of ϕ. The terms of the misclassification rate are the false negative and false positive rates, respectively.

The procedure presented in Algorithm 25 uses Algorithm 20 to compute the tightest deadlines for each trace. Then each deadline is determined in a greedy way such that the misclassification rate is minimized. The heuristic in Algorithm 20 is due to the fact that each deadline is considered separately from the others. However, the deadlines are not independent with respect to the minimization of the misclassification rate.

Notice that the algorithm constructs $\mathcal{A}_\infty$ only once at line 1. Then the automaton is used in the $tr(\cdot)$ function to compute the temporal relaxation of each trace, lines 2-3. Thus, the procedure avoids building $\mathcal{A}_\infty$ for each trace.

In Algorithm 25, m denotes the number of *within* operators and $\mathbf{b}$ is the m-dimensional vector of deadlines associated with each *within* operator in the TWTL formula ϕ. We assume that the order of the *within* operators is given by the post-order traversal of $AST(\phi)$, i.e., recursively traversing the children nodes first and then the node itself.

The complexity of the learning procedure is $O\big(2^{|\phi|+|AP|} + (|\mathcal{L}_p| + |\mathcal{L}_n|) \cdot l_m \cdot |\phi| + m^2 \cdot (|\mathcal{L}_p| + |\mathcal{L}_n|)\big)$, where: (a) the first term is the complexity of constructing $\mathcal{A}_\infty$ (line 1); (b) the second term corresponds to computing the tight deadlines for all traces positive and negative in lines 2 and 3, respectively; (c) the third term is the complexity of the for loop, which computes each deadline separately in a greedy fashion (lines

Algorithm 25: Parameter learning

Input: $\mathcal{L}_p$ – set of positive traces
Input: $\mathcal{L}_n$ – set of negative traces
Input: ϕ – template TWTL formula
Output: d – the vector of deadlines

1 $\mathcal{A}_\infty \leftarrow translate(\phi; \inf\ = \top)$
2 $D_p \leftarrow \{tr(p, \mathcal{A}_\infty) + \mathbf{b} \mid p \in \mathcal{L}_p\}$
3 $D_n \leftarrow \{tr(p, \mathcal{A}_\infty) + \mathbf{b} \mid p \in \mathcal{L}_n\}$
4 $\mathbf{d} \leftarrow (-\infty, -\infty, \ldots, -\infty)$ `//` m`-dimensional`
5 **for** $k \in \{1, \ldots, m\}$ **do**
6 $\quad D_k \leftarrow \{\mathbf{d}'[k] \mid \mathbf{d}' \in D_p \cup D_n\}$
7 $\quad \mathbf{d}[k] \leftarrow \arg\min_{d \in D_k} \left(\left|D^k_{FP}(d)\right| + \left|D^k_{FN}(d)\right| \right)$, where
8 $\quad\quad D^k_{FP}(d) \leftarrow \{\mathbf{d}'[k] \mid \mathbf{d}'[k] > d, \mathbf{d}' \in D_n\}$
9 $\quad\quad D^k_{FN}(d) \leftarrow \{\mathbf{d}'[k] \mid \mathbf{d}'[k] \leq d, \mathbf{d}' \in D_p\}$
10 **return** d

5-9). The maximum length of a trace (positive or negative) is denoted by l_m in the complexity formula.

5.8 TWTL Python Package

We provide a Python 2.7 implementation named PyTWTL of the proposed algorithms based on LOMAP (Ulusoy et al., 2013c), ANTLRv3 (Parr, 2007) and networkx (Hagberg et al., 2008) libraries. PyTWTL implementation is released under the GPLv3 license and can be downloaded from *hyness.bu.edu/twtl*. The library can be used to:

1. construct a DFA $\mathcal{A}_\phi$ and a annotated DFA $\mathcal{A}_\infty$ from a TWTL formula ϕ;
2. monitor the satisfaction of a TWTL formula ϕ;
3. monitor the satisfaction of an arbitrary relaxation of ϕ, i.e., $\phi(\infty)$;
4. compute the temporal relaxation of a trace with respect to a TWTL formula;
5. compute a satisfying control policy with respect to a TWTL formula ϕ;

6. compute a minimally relaxed control policy with respect to a TWTL formula ϕ, i.e., for $\phi(\boldsymbol{\tau})$ such that $|\boldsymbol{\tau}|_{TR}$ is minimal;

7. verify if all traces of a system satisfy some relaxed version of a TWTL formula ϕ;

8. learn the parameters of a TWTL formula ϕ, i.e., the deadlines of the *within* operators in ϕ.

The parsing of TWTL formulae is performed using ANTLRv3 framework. We provide grammar files which may be used to generate lexers and parsers for other programming languages such as Java, C/C++, Ruby. To support Python 2.7, we used version 3.1.3 of ANTLRv3 and the corresponding Python runtime ANTLR library, which we included in our distribution for convenience.

5.9 Case Studies

In this section, we present some examples highlighting the solutions for the verification, synthesis and learning problems. First, we show the automaton construction procedure on a TWTL formula and how the tight deadlines are inferred for a given trace. Then, we consider an example involving a robot whose motion is modeled as a TS. The policy computation algorithm is used to solve a path planning problem with rich specifications given as TWTL formulae. The procedure for performing verification, i.e., all robot trajectories satisfy a given TWTL specification, is also shown. Finally, the performance of the heuristic learning algorithm is demonstrated on a simple example.

5.9.1 Automata Construction and Temporal Relaxation

Consider the following TWTL specification over the set of atomic propositions $AP = \{A, B, C, D\}$:

$$\phi = [H^2 A]^{[0,6]} \cdot ([H^1 B]^{[0,3]} \vee [H^1 C]^{[1,4]}) \cdot [H^1 D]^{[0,6]} \tag{5.19}$$

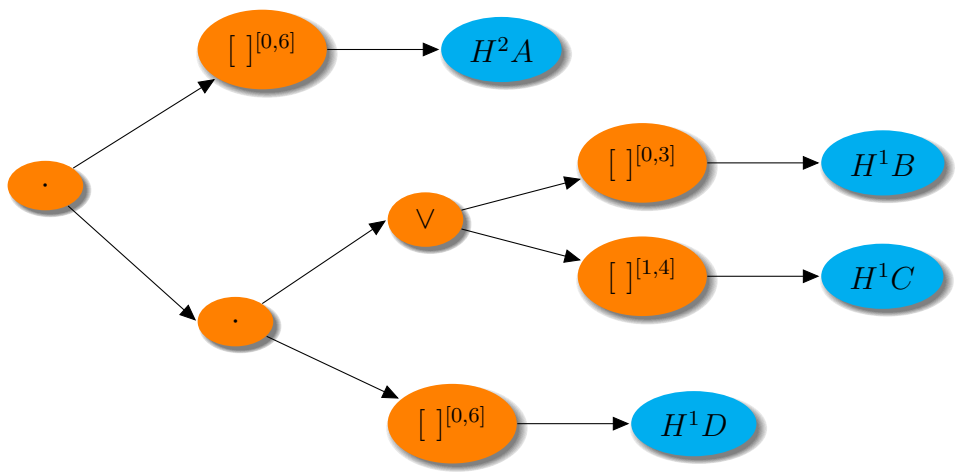
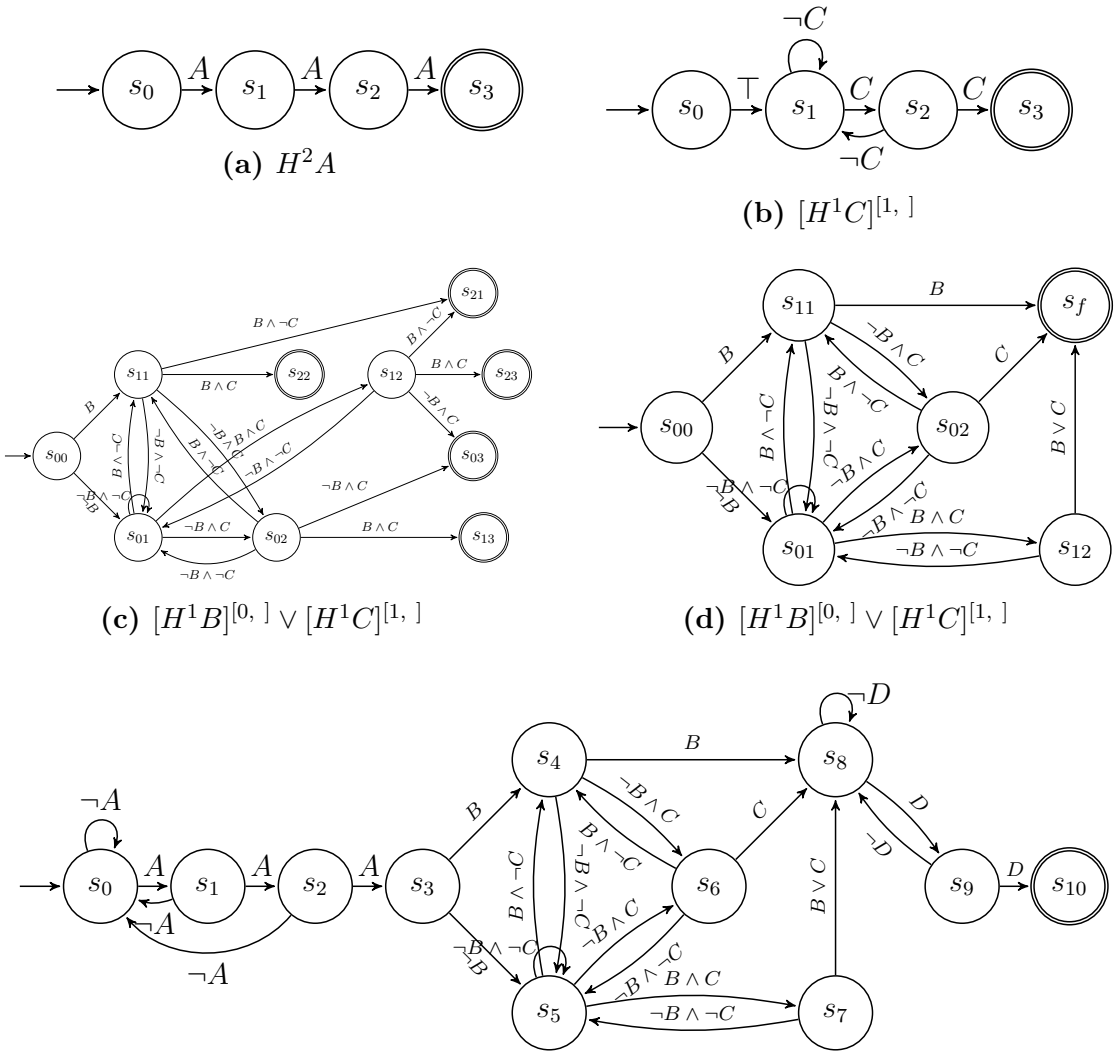


Figure 5·2: The AST corresponding to the TWTL formula in (5.19).

An AST of formula ϕ is shown in Figure 5·2. The TWTL formula ϕ is converted to an annotated DFA $\mathcal{A}_\infty$ using Algorithm 10. The procedure recursively constructs the DFA from the leafs of the AST to the root. A few processing steps are shown in Figure 5·3. The construction of DFA corresponding to a leaf, i.e., a *hold* operator, is straightforward, see Figure 5·3a. Next, the transformation corresponding to a *within* operator is shown in Figure 5·3b. Note that the delay of one time unit is due to the lower bound of the time window of the *within* operator. Also, note that the automaton restarts on symbols that block the DFA corresponding to the inner formula $H^1 C$.

The next two figures, Figure 5·3c and Figure 5·3d, show the translation of the disjunction operator. Specifically, Figure 5·3c, shows the product DFA corresponding to the disjunction without merging the final states. Since none of the final states have



(a) $H^2 A$

(b) $[H^1 C]^{[1,\ ]}$

(c) $[H^1 B]^{[0,\ ]} \vee [H^1 C]^{[1,\ ]}$

(d) $[H^1 B]^{[0,\ ]} \vee [H^1 C]^{[1,\ ]}$

(e) $[H^2 A]^{[0,\ ]} \cdot ([H^1 B]^{[0,\ ]} \vee [H^1 C]^{[1,\ ]}) \cdot [H^1 D]^{[0,\ ]}$

Figure 5·3: Annotated automata corresponding to subformulae of the TWTL specification in (5.19).

outgoing transitions, see Corollary 5.12, and they can be merged into a single final state, see Figure 5·3d. However, we still need to keep track of which subformula of the disjunctions holds. The annotation variable $T.choices$, introduced in Section 10, stores this information as

$$\begin{cases} L = \{(s_{11}, B \wedge \neg C), (s_{11}, B \wedge C), (s_{12}, B \wedge \neg C)\}, \\ R = \{(s_{02}, \neg B \wedge C), (s_{02}, B \wedge C), (s_{12}, \neg B \wedge C)\}, \\ B = \{(s_{12}, B \wedge C)\}. \end{cases} \tag{5.20}$$

Notice that the tuples in (5.20) correspond to the ingoing edges of the final states in the DFA from Figure 5·3c. Finally, the DFA corresponding to the overall specification formula ϕ is shown in Figure 5·3e.

Let $\phi_A = [H^2 A]^{[0,6]}$, $\phi_B = [H^1 B]^{[0,3]}$, $\phi_C = [H^1 C]^{[1,4]}$, and $\phi_D = [H^1 D]^{[0,6]}$ be subformulae of ϕ associated with the *within* operators. The annotation data for these subformulae is shown in the following table.

Subformula	$T.I$	$T.F$
ϕ	$\{s_0\}$	$\{s_{10}\}$
ϕ_A	$\{s_0\}$	$\{s_3\}$
ϕ_B	$\{s_3, s_5, s_6\}$	$\{s_8\}$
ϕ_C	$\{s_3\}$	$\{s_3\}$
ϕ_D	$\{s_8\}$	$\{s_{10}\}$

Consider the following word over the alphabet $\Sigma = 2^{AP}$:

$$\boldsymbol{\sigma} = \epsilon, \{A\}, \{A\}, \{A\}, \epsilon, \{B, C\}, \{B, C\}, \epsilon, \{D\}, \{D\} \tag{5.21}$$

where ϵ is the empty symbol. The following table shows the stages of Algorithm 20 as the symbols of the word $\boldsymbol{\sigma}$ are processed:

No.	Symbol	State	ϕ_A	ϕ_B	ϕ_C	ϕ_D
Init		s_0	$(\top, \bot, 0)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
0	ϵ	s_0	$(\top, \bot, 1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
1	$\{A\}$	s_1	$(\top, \bot, 2)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
2	$\{A\}$	s_2	$(\top, \bot, 3)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$	$(\bot, \bot, -1)$
3	$\{A\}$	s_3	$(\bot, \top, 3)$	$(\top, \bot, 0)$	$(\top, \bot, 0)$	$(\bot, \bot, -1)$
4	ϵ	s_5	$(\bot, \top, 3)$	$(\top, \bot, 1)$	$(\top, \bot, 1)$	$(\bot, \bot, -1)$
5	$\{B, C\}$	s_7	$(\bot, \top, 3)$	$(\top, \bot, 2)$	$(\top, \bot, 2)$	$(\bot, \bot, -1)$
6	$\{B, C\}$	s_8	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\top, \bot, 0)$
7	ϵ	s_8	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\top, \bot, 1)$
8	$\{D\}$	s_9	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\top, \bot, 2)$
9	$\{D\}$	s_{10}	$(\bot, \top, 3)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$	$(\bot, \top, 2)$

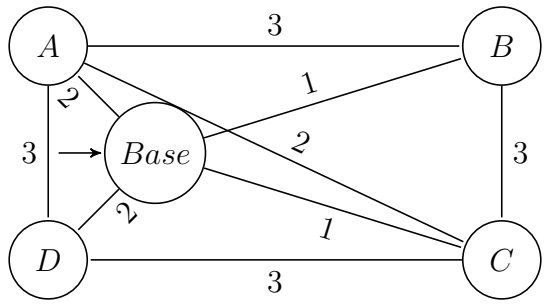
where each 3-tuple in last four columns represents the annotation variables $T.ongoing$, $T.done$ and $T.steps$, respectively. The temporal relaxation for $\boldsymbol{\sigma}$ can be extracted from the values in the last row by subtracting the deadlines of the *within* operators from them. Thus, the vector of tightest τ values is $(-3, -1, -2, -3)$. However, because ϕ_B and ϕ_C are in disjunction, we have the temporal relaxation $\boldsymbol{\tau} = (-3, -\infty, -2, -3)$, where we choose to ignore the subformula containing ϕ_B. Thus, the maximum temporal relaxation is $|\boldsymbol{\tau}|_{TR} = -2$.

5.9.2 Control Policy Synthesis

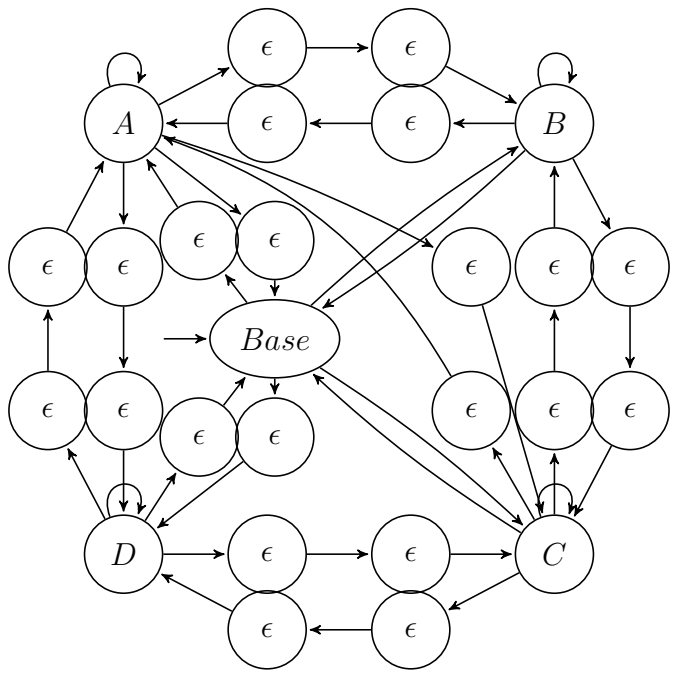
Consider a robot moving in an environment represented as the finite graph shown in Figure 5·4a. The nodes of the graph represent the points of interest, while the edges indicate the possibility of moving the robot between the edges' endpoints. The numbers associated with the edges represent the travel times, and we assume that all the travel times are integer multiples of a time step Δt. The robot may also stay at any of the points of interest.

The motion of the robot is abstracted as a transition system $\mathcal{T}$, which is obtained from the finite graph by splitting each edge into a number of transitions equal to the corresponding edge's travel time. The generated transition system thus has 27 states

and 67 transitions and is shown in Figure 5·4b.



(a) An environment with five points of interest, *Base* A, B, C, and D. The edges indicate the existence of paths between their endpoints, while the associated numbers represent the travel times of the edges. The robot may stay at a region of interest.



(b) The transition system $\mathcal{T}$ obtained from the environment graph shown in Figure 5·4a.

Figure 5·4: The environment where the robot operates and its abstraction $\mathcal{T}$.

Consider the TWTL specification ϕ from (5.19). The product automaton $\mathcal{P} = \mathcal{T} \times \mathcal{A}_\infty$ is constructed, where $\mathcal{A}_\infty$ is the annotated DFA corresponding to $\phi(\infty)$ shown in Figure 5·3e. The product automaton $\mathcal{P}$ has 204 states and 378 transitions.

The control policy computed by using Algorithm 23 is

$$\mathbf{x} = Base, A, A, A, C, C, Base, D, D, \tag{5.22}$$

which generates the output word

$$\boldsymbol{\sigma} = \epsilon, \epsilon, \{A\}, \{A\}, \{A\}, \epsilon, \{C\}, \{C\}, \epsilon, \epsilon, \{D\}, \{D\}. \tag{5.23}$$

The minimum temporal relaxation for $\boldsymbol{\sigma}$ is $|\boldsymbol{\tau}|_{TR} = -2$, where $\boldsymbol{\tau} = (-2, -\infty, -2, -3)$ is the minimal temporal relaxation vector associated with $\boldsymbol{\sigma}$.

5.9.3 Verification

In the verification problem, we are concerned with checking for the existence of relaxed specifications for every possible run of a transition system.

To illustrate this problem, consider the transition system in Figure 5·5 and the following two TWTL specifications:

$$\phi_1 = [H^1 A]^{[1,2]} \tag{5.24}$$

$$\phi_2 = [H^3 \neg B]^{[1,2]} \tag{5.25}$$

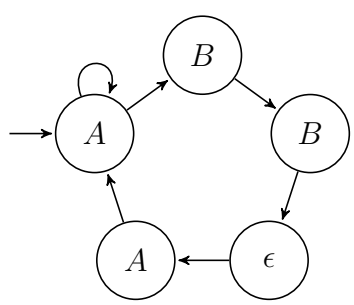


Figure 5·5: A simple transition system $\mathcal{T}^{simple}$.

To check the transition system $\mathcal{T}^{simple}$ against the two specifcations, we can use Algorithm 24. It is straightforward that the procedure will return true for ϕ_1, because

every run of $\mathcal{T}^{simple}$ satisfies $\phi_1(3) = [H^1 A]^{[1,2+3]}$. Note that the runs of the transition system may not need to satisfy the original specification as the satisfaction of a relaxed version is sufficient. Similarly, Algorithm 24 returns false for ϕ_2, because there exists a run of $\mathcal{T}^{simple}$ that does not satisfy any relaxation of the specification, e.g., $\mathbf{x} = (A, B, B, \epsilon, A)^*$.

An important conclusion highlighted by the two examples is that the verification problem proposed in this paper is concerned with checking a system against the logical structure of a specification and not against any particular time bounds. This might be useful in situation where the deadlines of the specification are not known *a priori*, but the logical structure of the specification is.

5.9.4 Learning deadlines from data

In the previous two cases, we use the TWTL specifications in conjunction with problems involving infinite sets of words encoded as transition systems. However, it is often the case that only finite sets of (output) trajectories are available. In this section, we give a simple example of the learning problem presented in Section 5.4.

Consider the specification $\phi_l = [H^1 A]^{[0,d_1]} \cdot [H^2 B]^{[0,d_2]}$ with unknown deadlines and the following set of labeled trajectories, where C_p and C_n are the positive and negative example labels, respectively:

Word	Label	Deadlines
$\boldsymbol{\sigma}_1 = \{A\},\{A\},\{A\},\{B\},\{B\},\{B\},\{B\},\epsilon$	C_p	$(1,3)$
$\boldsymbol{\sigma}_2 = \epsilon,\ \{A\},\{A\},\epsilon,\ \{B\},\{B\},\{B\},\epsilon$	C_p	$(2,3)$
$\boldsymbol{\sigma}_3 = \{B\},\epsilon,\ \{A\},\{A\},\{B\},\{B\},\{B\},\{B\}$	C_n	$(3,2)$
$\boldsymbol{\sigma}_4 = \epsilon,\ \{A\},\{A\},\epsilon,\ \epsilon,\ \{B\},\{B\},\{B\}$	C_n	$(2,4)$

The last column in the above table shows the tight deadlines obtained in lines 2 and 3 of Algorithm 25. Next, the learning algorithm computes the heuristic sets D^k_{FP} and D^k_{FN}, $k \in \{d_1, d_2\}$, of false positive and false negative trajectories, respectively:

Deadline	Value	D^k_{FP}	D^k_{FN}	$\left\lvert D^k_{FP}\right\rvert + \left\lvert D^k_{FN}\right\rvert$
d_1	1	$\emptyset$	$\{\boldsymbol{\sigma}_2\}$	1
d_1	2	$\{\boldsymbol{\sigma}_4\}$	$\emptyset$	1
d_1	3	$\{\boldsymbol{\sigma}_3, \boldsymbol{\sigma}_4\}$	$\emptyset$	2
d_2	2	$\{\boldsymbol{\sigma}_3\}$	$\{\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2\}$	3
d_2	3	$\{\boldsymbol{\sigma}_3\}$	$\emptyset$	1
d_2	4	$\{\boldsymbol{\sigma}_3, \boldsymbol{\sigma}_4\}$	$\emptyset$	2

Finally, Algorithm 25 chooses the deadline pair $\mathbf{d} = (d_1, d_2) = (1, 3)$ that has the lowest heuristic misclassification rate, $\left\lvert D^k_{FP}\right\rvert + \left\lvert D^k_{FN}\right\rvert$ shown in the last column of the above table, for d_1 and d_2, respectively. An important observation is that the inferred formula $\phi_l^{\mathbf{d}} = [H^1 A]^{[0,1]} \cdot [H^2 B]^{[0,3]}$ has zero as actual misclassification rate. The discrepancy between the values in the table and the actual value of the final misclassification rate are due to the heuristic of synthesizing each deadline separately. Thus, the heuristic procedure in Algorithm 25 ignores the temporal and logical structure of the template TWTL formula which may lead to suboptimal performance, i.e., misclassification rate.

We also tested the learning algorithm on larger sets of trajectories. Algorithm 25 was ran using the template TWTL formula $[H^2 A]^{[0,d_1]} \cdot [H^3 B]^{[2,d_2]} \cdot [H^2 C]^{[0,d_3]}$. The inference was performed using a set of 100 trajectories, 50 positive and 50 negative, shown in Figure 5·6. Executing Algorithm 25 returned the vector of deadlines $(d_1, d_2, d_3) = (29, 40, 31)$ that induces a misclassification rate of 14%.

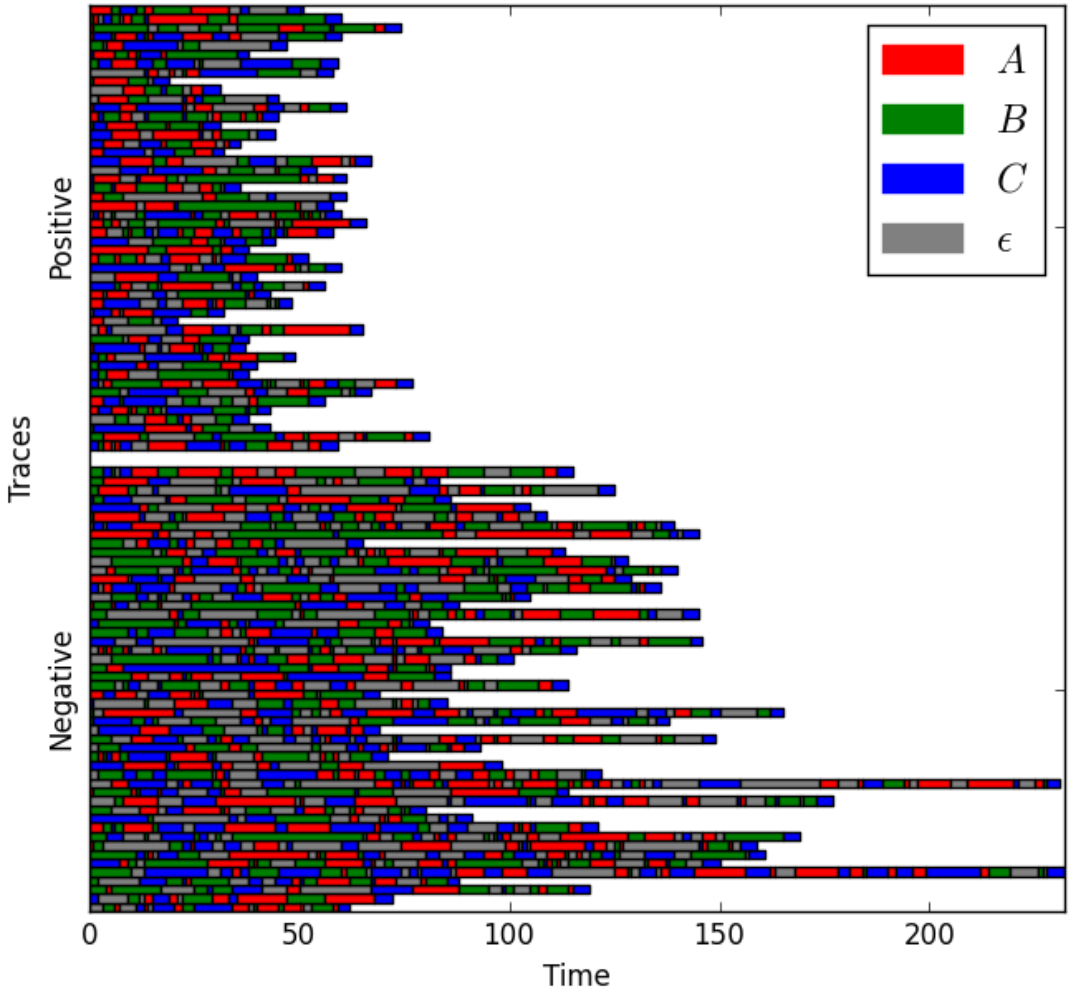


Figure 5·6: The training set contains 50 positive and 50 negative labeled trajectories.

Chapter 6

Persistent Vehicle Routing Problem with Charging and Temporal Logic Constraints

We propose a new formulation and algorithms for the Vehicle Routing Problem (VRP). To accommodate persistent surveillance missions, which require executions in infinite time, we define Persistent VRP (P-VRP). The vehicles consume a resource, such as gas or battery charge, which can be replenished when they visit replenish stations. The mission specifications are given as rich, temporal logic statements about the sites, their service durations, and the time intervals in which services should be provided. Two different optimization criteria are considered. The first is the infinite-time limit of the duration needed for the completion of a surveillance round. The second penalizes the long-term average of the same quantity. The proposed algorithms, which are based on concepts and tools from formal verification and optimization, generate collision-free motion plans automatically from the temporal logic statements and vehicle characteristics such as maximum operation time and minimum replenish time. Illustrative simulations and experimental trials for a team of quadrotors involved in persistent surveillance missions are included.

6.1 Environment and Vehicle Models

For simplicity of presentation, we assume the team is made of N identical vehicles. At the end of the paper, we discuss how this assumption can be relaxed. Let $\mathcal{E} =$

$(Q = \mathcal{S} \cup \mathcal{C}, \Delta, \varpi)$ be a graph environment, where $\mathcal{S}$ is the set of sites and $\mathcal{C}$ is the set of replenish stations or depos. An edge $e \in \Delta \subseteq Q \times Q$ denotes that a vehicle can move between the source and destination of the edge. We assume that the vehicles can deterministically choose to traverse the edges of $\mathcal{E}$, stay at a site for service, or stay docked in a charging station. Each edge has an associated duration given by $\varpi : \Delta \to \mathbb{Z}_{\geq 1}$. We assume that the duration associated to an edge includes the time for obstacle avoidance maneuvers and docking or undocking, if applicable. For now, we assume that this value is the exact time that a vehicle needs to travel the corresponding edge. However, the method developed in this paper also works for the case when this value is an upper bound for the travel time.

We assume that a collision between two vehicles can occur in one of the following three situations: (1) both are at the same node at the same time; (2) both traverse the same edge at the same time (they may start the motion at different times); (3) a vehicle arrives at a node less that t_{col} after the departure of another vehicle from the same node.

Each vehicle has a limited amount of a resource, such as fuel or battery charge, and must regularly return to a replenish station. For simplicity, we assume that the resource is battery charge (level), and we will refer to the replenish stations as charging stations. We use t_{op} to denote the maximum operation time for a vehicle starting with a fully charged battery and t_{ch} to denote the charging time starting with an empty battery. For simplicity, we assume that time is discretized, and all durations (e.g., $\varpi(\Delta)$, t_{col}, t_{op}, t_{ch}) are expressed as an integer multiple of a time interval Δt. Let $\gamma = \frac{t_{ch}}{t_{op}} \geq 1$ be the charge-discharge (integer) ratio.

The *battery state* $b_t(i)$ of vehicle $i \in \{1, \ldots, N\}$ at time $t \in \mathbb{Z}_{\geq 0}$ is discretized such that $b_t(i) \in \{0, \ldots, t_{ch}\}$. The update rule for $b_t(i)$ after d time units is defined as

follows:

$$b_{t+d}(i) = \begin{cases} \min\{b_t(i) + d, t_{ch}\} & \text{vehicle } i \text{ is docked} \\ b_t(i) - \gamma d & \text{otherwise} \end{cases} \tag{6.1}$$

The batteries may be charged at any of the charging stations $\mathcal{C}$. Charging may start and stop at any battery state. Once a vehicle is fully charged, it will remain fully charged until it leaves the charging station. We assume that at the start of the mission all vehicles are fully charged and docked.

At each time, each vehicle may be in one of the following four states: (1) *moving* between sites and charging stations, (2) *servicing* a request at a site, (3) *charging* or (4) *idle* if docked and fully charged. If a vehicle is either moving or servicing a request, we will say that the vehicle is *active*. A time interval such that all vehicles are docked and at least one is charging is called *no flight time* (NFT). A time interval in which all vehicles are idle is called *idle time*. We require that NFTs and idle times are maximal time intervals, i.e. they may not be extended on either side while maintaining their defining property.

For $q \in Q$, we use $\vec{q}$ to denote that a vehicle is moving towards q. Let $\vec{Q} = \{\vec{q} \,|\, q \in Q\}$. A *control policy* for the N vehicle system is a sequence $\mathbf{v} = v_1 v_2 \ldots$ where $v_t \in (Q \cup \vec{Q})^N$ specifies at each time $t \in \mathbb{Z}_{\geq 0}$ and for each vehicle $i \in \{1, \ldots, N\}$ if vehicle i is at a site or charging station or if it is moving. Let $v_t(i)$ and $v(i)$, $i \in \{1, \ldots, N\}$, denote the control value for vehicle i at time t and the control policy for vehicle i (i.e., the sequence of control values), respectively. Then a transition $(q_1, q_2) \in \Delta$ performed by vehicle i starting at time t will correspond to $v_t(i) = q_1$, $v_{t+d}(i) = q_2$ and $v_{t+k}(i) = \vec{q}_2$, $k \in \{1, \ldots, d-1\}$, where $d = \varpi((q_1, q_2))$ is the duration of the transition. Servicing or charging for one time interval (Δt time) by vehicle i at time t corresponds to $v_t(i) = v_{t+1}(i) \in Q$.

For a control policy $\mathbf{v} = v_1 v_2 \ldots$ we define the corresponding output word $\mathbf{o} =$

$o_1 o_2 \ldots$, where $o_t = \{v_t(i) | v_t(i) \in \mathcal{S}, i \in \{1, \ldots, N\}\}$ is the set of all sites occupied by the N vehicles at time $t \in \mathbb{Z}_{\geq 0}$. We use ϵ to denote that no site is occupied. Let $q^{[d]}$ and q^{ω} denote d and infinitely many repetitions of q, respectively.

Example 6.1. *An example for the case of $N = 2$ vehicles, 3 sites, and 3 charging stations is shown in Figure 6·1. A possible control policy* $\mathbf{v}$ *for vehicle 1 (blue) and vehicle 2 (red) is:*

$$\begin{aligned} v(1) &= Ch_3^{[1]}\vec{C}^{[3]}C^{[4]}\vec{A}^{[2]}, A^{[3]}\vec{Ch}_1^{[3]}, Ch_1^{[18]}Ch_1^{[54]} \\ &\quad \left(Ch_1^{[1]}\vec{C}^{[3]}C^{[4]}\vec{A}^{[2]}A^{[3]}\vec{Ch}_1^{[3]}Ch_1^{[18]}Ch_1^{[54]}\right)^{\omega} \\ v(2) &= Ch_2^{[1]}Ch_2^{[17]}\vec{B}^{[3]}B^{[3]}\vec{C}^{[4]}C^{[3]}\vec{Ch}_3^{[4]}Ch_3^{[54]} \\ &\quad \left(Ch_3^{[1]}Ch_3^{[17]}\vec{B}^{[3]}, B^{[3]}\vec{C}^{[4]}C^{[3]}\vec{Ch}_3^{[4]}Ch_3^{[54]}\right)^{\omega} \end{aligned} \tag{6.2}$$

Under control strategy (6.2), the blue vehicle services sites C and A and the red vehicle services sites B and A infinitely often. The blue and red vehicles always return to Ch_1 and Ch_3, respectively. The corresponding output word is

$$o = \left(\epsilon^{[4]}C^{[4]}\epsilon^{[2]}A^{[3]}\epsilon^{[8]}B^{[3]}\epsilon^{[4]}C^{[3]}\epsilon^{[58]}\right)^{\omega}.$$

6.2 P-VRP Formulation

Let $\mathbf{v}$ be a control policy. We say that $\mathbf{v}$ is *feasible* if at each moment in time the N vehicles are pairwise in collision free states and have non-negative battery states, i.e., $b_t(i) \geq 0$ for all $i \in \{1, \ldots, N\}$ and $t \in \mathbb{Z}_{\geq 0}$.

Definition 6.1 (Persistent surveillance)**.** *A control policy is said to satisfy the persistent surveillance specification $\mathbb{G}\phi$, where ϕ is a TWTL formula, if the generated output word satisfies the TWTL formula ϕ infinitely often and there is no idle time between any two consecutive satisfactions of ϕ.*

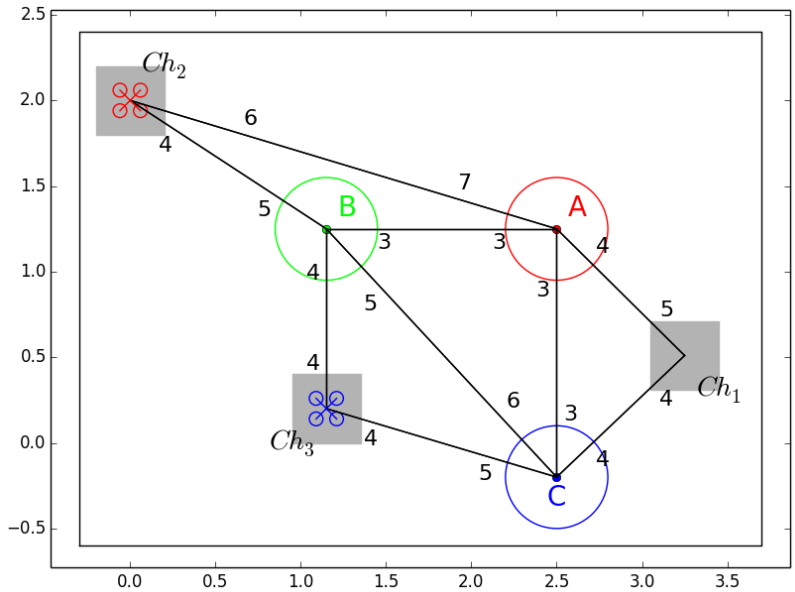


Figure 6·1: An environment with 3 sites, $\mathcal{S} = \{A, B, C\}$, and 3 charging stations, $\mathcal{C} = \{Ch_1, Ch_2, Ch_3\}$. The two numbers associated to each edge correspond to the durations for the two directions of the edge, e.g., the durations of edges (B, C) and (C, B) are 5 and 6, respectively. The vehicles, shown in blue and red, start fully charged from the charging stations Ch_3 and Ch_2, respectively. The charging time is $t_{ch} = 60$, the operation time is $t_{op} = 20$ ($\gamma = 3$), and the collision time is $t_{col} = 2$.

Note that, while the satisfaction of $\mathbb{G}\phi$ does not allow for idle time between successive satisfactions of ϕ, there may be no flight time to allow for the vehicles to recharge.

Problem 6.1 (P-VRP Completeness)**.** *Given an environment $\mathcal{E} = (Q = \mathcal{S} \cup \mathcal{C}, \Delta, \varpi)$, N vehicles, operation time t_{op}, charging time t_{ch}, collision time t_{col}, and a TWTL formula ϕ over $\mathcal{S}$, find a feasible control policy that satisfies $\mathbb{G}\phi$ if one exists, otherwise report failure.*

Let $\mathbf{v}$ be a feasible control policy satisfying $\mathbb{G}\phi$. We define a *loop* as a finite subsequence of $\mathbf{v}$ starting with the satisfaction of the formula ϕ and ending before the next satisfaction.

Example 6.2 (Example 6.1 revisited)**.** *Consider a mission in which it is required to service site A for 2 time units within* $[0, 12]$ *and site C for 3 time units within* $[0, 9]$*. In addition, within* $[0, 32]$*, site B needs to be serviced for 2 time units followed by either A or C for 2 time units within* $[0, 8]$*. All the above requirements need to be satisfied infinitely often. The corresponding formula is* $\mathbb{G}\phi_{tw}$*, where*

$$\phi_{tw} = [H^2 A]^{[0,12]} \wedge [H^2 B[H^2 A \vee C]^{[0,8]}]^{[0,32]} \wedge [H^3 C]^{[0,9]}$$

The control policy from (6.2) *satisfies the above persistent surveillance specification. It is easy to note that it is also feasible because at most one vehicle is active at all times and the battery states for both vehicles are always non-negative. For each vehicle, the control policy has a loop ending after a NFT, which is marked in gray. The NFTs ensure that the vehicles start each loop fully charged.*

Let $T(k)$ be the start time of the k-th loop and $\Delta T(k) = T(k+1) - T(k)$ be the *loop time.* Let

$$J_1(\mathbf{v}) = \limsup_{k\to\infty} \Delta T(k) \tag{6.3}$$

and

$$J_2(\mathbf{v}) = \lim_{k\to\infty} \frac{T(k+1)}{k} = \lim_{k\to\infty} \frac{\sum_{i=1}^{k} \Delta T(i)}{k} \tag{6.4}$$

be two cost functions that penalize the asymptotic upper bound of the loop time and the long-term average loop time, respectively.

Problem 6.2 (Optimality)**.** *Under the same assumptions stated in Problem 6.1, find a satisfying and feasible control policy that minimizes* J_1 *(or* J_2*) if one exists, otherwise report failure.*

Our approach to Problems 6.1 and 6.2 is inspired from automata-based model checking. The TWTL formula is translated to a finite state automaton that accepts

the satisfying language. This is then composed with finite transition systems modeling the motion of the vehicles in the environment and the charging constraints. The satisfiability and optimality problems are solved on the resulting product automaton.

6.3 Control Policy

For a finite set Σ, we denote by $P_k(\Sigma)$ the set of k-permutations.

Algorithm 26: Product Automaton

Input: $\mathcal{T}$ – transition system
Input: ϕ – specification as a TWTL formula
Input: N – number of vehicles
Input: t_{col} – collision time
Input: t_{op}, t_{ch} – operation time and charging time

1 Construct product transition system $\mathcal{T}^N$ for the ME or FC operation mode
2 Generate charging FSA $\mathcal{A}^{ch}$ with t_{op}, t_{ch} for N vehicles
3 Construct product $\mathcal{P}^{ch} = \mathcal{T}^N \times \mathcal{A}^{ch}$
4 Transform ϕ to an $scLTL$ formula ψ
5 Construct the FSA $\mathcal{A}^{spec}$ corresponding to ψ
6 Construct product automaton $\mathcal{P} = \mathcal{P}^{ch} \times \mathcal{A}^{spec}$

6.3.1 Motion model

We consider two modes of operation: (1) mutually exclusive mode, which assumes that at any given time at most one vehicle is active (i.e., moving or servicing a request), and (2) fully concurrent mode, which does not place any restrictions except that the vehicles must be in collision free states at all times. The mutually exclusive mode of operation has the advantage that it guarantees collision free control policies and also extended overall operation time for the vehicles. This is a good fit for surveillance missions, but may not be desired for rescue missions, where a parallel search approach may be more effective. Also, as discussed at the end of the section, the complexity of the presented algorithms is lower for the mutually exclusive mode.

Algorithm 27: Planning Algorithm – completeness

Input: $\mathcal{T}$ – transition system
Input: ϕ – specification as a TWTL formula
Input: N – number of vehicles
Input: t_{col} – collision time
Input: $q_0 \in P_N(\mathcal{C})$ – initial vehicle locations
Input: t_{op}, t_{ch} – operation time and charging time
Output: $\mathbf{v}$ – control policy

1 $\mathcal{P} \leftarrow ConstructPA(\mathcal{T}, \phi, N, t_{col}, t_{op}, t_{ch})$
2 $G = (V, E)$, $V = P_N(\mathcal{C})$, $E = \emptyset$
3 **for** $(q_1, q_2) \in V \times V$ **do**
4 if there is a satisfying path in $\mathcal{P}$ starting fully charged in q_1 and ending fully charged in q_2 then $E \leftarrow E \cup (q_1, q_2)$ and $control_{\mathcal{P}}(q_1, q_2)$ stores the computed path in $\mathcal{P}$
5 **if** *G is acyclic or no cycle is reachable from q_0* **then**
6 **return** Failure
7 **else**
8 find a cycle q_c and a path q_p to the cycle in G
9 **return** $\beta_{\mathcal{T}^N}(control_{\mathcal{P}}(q_p)(control_{\mathcal{P}}(q_c))^{\omega})$

The motion of a single vehicle is captured by a weighted transition system $\mathcal{T} = (Q, q_0, \bar{\Delta}, \bar{\varpi}, \Pi, h)$, where $Q = \mathcal{S} \cup \mathcal{C}$ is the set of states, $q_0 \in \mathcal{C}$ is the initial state, $\bar{\Delta} = \Delta \cup \{(q, q) | q \in Q\}$ is the set of transitions, $\bar{\varpi} : \bar{\Delta} \to \mathbb{Z}_{\geq 1}$ is the weight function, $\Pi = \mathcal{S} \cup \{\epsilon\}$ is the alphabet, and $h : Q \to \Pi$ is the labeling function. The weights represent the durations of transitions such that $\bar{\varpi}(q, q') = \varpi(q, q')$, for $q \neq q'$, and $\bar{\varpi}(q, q') = 1$, for $q = q'$. Thus, servicing and docking are modeled as self-loop transitions with duration 1. The labeling function only assigns values to sites, i.e. $h(q) = q$ for $q \in \mathcal{S}$ and $h(q) = \epsilon$ otherwise.

Mutually exclusive (ME) operation mode

In order to capture the motion of all vehicles at the same time, we define a mutually-exclusive product transition system (PTS) as a tuple $\mathcal{T}^N = (\tilde{Q}, \tilde{q}_0, \tilde{\Delta}, \tilde{\omega}, \Pi^N, \tilde{h})$. The set of states is defined such that there is at most one active vehicle, $\tilde{Q} =$

$P_N(\mathcal{C}) \cup (\bigcup_{k=1}^{N}\{(q_1,\ldots,q_N)|q_k \in \mathcal{S}, (q_1,\ldots,q_{k-1},q_{k+1},\ldots,q_N) \in P_{N-1}(\mathcal{C})\})$. At the initial state, it is assumed that all vehicles are docked, $\tilde{q}_0 \in P_N(\mathcal{C})$. A transition $(q_1,\ldots,q_N) \to (q'_1,\ldots,q'_N) \in \tilde{\Delta}$ if: (1) $q_i = q'_i$, $\forall i$, or (2) $q_k \to q'_k \in \Delta$, $q_i = q'_i$, $\forall i \neq k$ and $q'_k \neq q_j$, $\forall j \neq k$. The weight of a transition is 1 if the two endpoints are the same or equal to the weight of the transition in Δ corresponding to the second case above. The labeling function is defined component-wise, $\tilde{h}(q_1,\ldots,q_N) = (h(q_1),\ldots,h(q_N))$.

Fully concurrent (FC) operation mode

We define a similar product transition system (PTS) $\mathcal{T}^N = (\tilde{Q}, \tilde{q}_0, \tilde{\Delta}, \tilde{\omega}, \Pi^N, \tilde{h})$ as before, but in this case we account for simultaneous active vehicles and collisions. The simultaneous motions of the vehicles lead to a synchronization problem. Due to space constraints, we only include an informal description of how this synchronization problem is solved. We split all the edges of the single-vehicle transition system $\mathcal{T}$ into edges of duration 1. We then proceed to compute the full PTS, which captures all possible motions of the N vehicles, using this modified transition system. The last step is to eliminate the states and edges of the PTS that determine collisions according to the description from Section 6.1.

Note that we achieve collision avoidance using temporal separation, instead of spatial separation as in the case of geometric approaches such as RRT (LaValle and Kuffner, 1999; LaValle, 2006) or PRM (Kavraki et al., 1996). In our case, this is beneficial since it prunes the PTS of undesired states and actually helps lower the computation time in the fully concurrent mode. Also, for the case of quadrotors, temporal separation also avoids undesired aerodynamic effects which may arise due to close proximity of the vehicles. One example is the loss of lift when a quadrotor is directly below another one. These phenomena are somewhat hard to encode in geometric approaches.

6.3.2 Charging model

The charging process is modeled as a Finite State Automaton (FSA). Recall that the charging time t_{ch} is an integer multiple of Δt and $\gamma = \frac{t_{ch}}{t_{op}} \in \mathbb{Z}_{\geq 1}$ (see Section 6.2). For the ME operation mode, the charging FSA is $\mathcal{A}^{ch} = (S^{ch}_{\mathcal{A}}, s^{ch}_0, \Sigma^{ch}, \delta^{ch}_{\mathcal{A}}, F^{ch}_{\mathcal{A}})$. $S^{ch}_{\mathcal{A}} = (\{0, \ldots, t_{ch}\})^N$ is the set of states. A state stores the battery states for all vehicles. The initial state is $s^{ch}_0 = (t_{ch}, \ldots, t_{ch})$ and corresponds to all vehicles being fully charged. The alphabet is $\Sigma^{ch} = (\{(i,0)|1 \leq i \leq N\} \cup \{(0,i)|1 \leq i \leq N\} \cup \{(i,i)|0 \leq i \leq N\}) \times D$, where D is the set of the durations of all transition of $\mathcal{T}$. Each triple represents the current and previous active vehicle and the duration of a transition from $\mathcal{T}$. The value 0 for the current or previous active fields indicates that no vehicle is active. By this convention, the three sets of pairs in the definition of the alphabet capture undocking, docking, and moving or servicing performed by vehicle i, respectively. The transition function is defined for two cases: (1) if all robots are recharging ($c = p = 0$), then

$$\delta^{ch}_{\mathcal{A}}((t_1, \ldots, t_N), (c, p, d)) = (min(t_1 + d, t_{ch}), \ldots, min(t_N + d, t_{ch}));$$

(2) if one robot is active ($c > 0$ or $p > 0$), then

$$\delta^{ch}_{\mathcal{A}}((t_1, \ldots, t_N), (c, p, d)) = (min(t_1 + d, t_{ch}), \ldots, t_a - \gamma d, \ldots, min(t_N + d, t_{ch})),$$

where $a = \max(c, p)$. Note that the transition function resembles the charging rule defined in Section 6.1. The set of final states $F^{ch}_{\mathcal{A}}$ can be the whole set of states $S^{ch}_{\mathcal{A}}$ if no restrictions on the final battery states are defined. However, we will impose some restrictions on $F^{ch}_{\mathcal{A}}$ later in this section.

For the FC operation mode, we modify the alphabet to $\Sigma^{ch} = \{0,1\}^N \times \{0,1\}^N$, which specifies for each vehicle if it was docked or active in the current and pre-

vious time steps, respectively. The transition function must also be adapted. Let $(t'_1, \ldots, t'_N) = \delta^{ch}_{\mathcal{A}}((t_1, \ldots, t_N), ((c_1, \ldots, c_N), (p_1, \ldots, p_N)))$. Then $t'_i = \min(t_i + 1, t_{ch})$ if $c_i = p_i = 0$, or $t'_i = t_i - \gamma$ otherwise, for all $i \in \{1, \ldots N\}$. Note that, in this case, we do not include the durations of transitions in the alphabet, because by construction all the transitions of the PTS have duration one in the FC mode.

6.3.3 Specification

To enforce the specification, we encode it as an automaton. We use the Algorithm 10 from Chapter 5 to obtain a FSA $\mathcal{A}^{spec}$ that accepts the language over $2^{\mathcal{S}}$ that satisfies the formula.

Another, less efficient, option is to first translate the TWTL formula into scLTL formula (Kupferman and Y. Vardi, 2001), and then use an off-the-shelf tool, such as *scheck* (Latvala, 2003). Note that the size of obtained scLTL formulae is of the order of the size of the TWTL formula times its time bound, thus yielding formulae which are too long to handle by an operator. Consider the specification is to "satisfy A for 2 time units before 10". The TWTL is $[H^2 A]^{[0,10]}$, while the corresponding scLTL formula has 24 operands and 75 operators for a total size of 99. Thus, a specification language such as TWTL, which incorporates time bounds in the operators and avoids the explosion of the formulae sizes, becomes necessity, rather than convenience.

6.3.4 Completeness

To provide a solution to Problem 6.1, we first define a product automaton that captures all feasible motions of the team that satisfy the specification and the charging constraints. First, we construct the product automaton $\mathcal{P}^{ch} = \mathcal{T}^N \times \mathcal{A}^{ch}$ between the motion model $\mathcal{T}^N$ and the charging FSA $\mathcal{A}^{ch}$. We define it as $\mathcal{P}^{ch} = (S^{ch}_{\mathcal{P}}, s^{ch}_{\mathcal{P}0}, \Delta^{ch}_{\mathcal{P}}, \omega^{ch}_{\mathcal{P}}, \Pi^N, h^{ch}_{\mathcal{P}})$, where $S^{ch}_{\mathcal{P}} = \tilde{Q} \times S^{ch}_{\mathcal{A}}$ is the set of states, $s^{ch}_{\mathcal{P}0} = (q_0, s^{ch}_0)$

is the initial state, $\Delta_{\mathcal{P}}^{ch} \subseteq \tilde{\Delta} \times \delta_{\mathcal{A}}^{ch}$ is the transition function, $\omega_{\mathcal{P}}^{ch} : \Delta_{\mathcal{P}}^{ch} \to \mathbb{Z}_{\geq 1}$ is the weight function, and $h_{\mathcal{P}}^{ch} : S_{\mathcal{P}}^{ch} \to \Pi^N$ is the labeling function. $\omega_{\mathcal{P}}^{ch}$ and $h_{\mathcal{P}}^{ch}$ are inherited from the transition system $\mathcal{T}^N$, i.e. for all $(q,t) \in S_{\mathcal{P}}^{ch}$ and $((q,t),(q',t')) \in \Delta_{\mathcal{P}}^{ch}$ we have $\omega_{\mathcal{P}}^{ch}((q,t),(q',t')) = \tilde{\omega}(q,q')$ and $h_{\mathcal{P}}^{ch}(q,t) = \tilde{h}(q)$.

In the ME mode, a transition $(\tilde{q},t) \to (\tilde{q'},t')$ is in $\Delta_{\mathcal{P}}^{ch}$ if $\tilde{q} \to \tilde{q'} \in \tilde{\Delta}$, $t \to^{(c,p,d)} t'$, $d = \tilde{\omega}(q,q')$ and c and p are the indices of the active vehicle in states q' and q, respectively. If all vehicles are charging at state q or q', then $p = 0$ or $c = 0$, respectively. For the FC mode, a transition $(\tilde{q},t) \to (\tilde{q'},t')$ is in $\Delta_{\mathcal{P}}^{ch}$ if $\tilde{q} \to \tilde{q'} \in \tilde{\Delta}$ and $t \to^{(c,p)} t'$, where $c = (c_1,\ldots,c_N)$ and $p = (p_1,\ldots,p_N)$ specify for each vehicle $i \in \{1,\ldots N\}$ if it is active in states q' and q, respectively.

Second, we construct the product automaton $\mathcal{P} = \mathcal{P}^{ch} \times \mathcal{A}^{spec}$. This is defined as $\mathcal{P} = (S_{\mathcal{P}}, s_0, \delta_{\mathcal{P}}, \omega_{\mathcal{P}}, F_{\mathcal{P}})$, where $S_{\mathcal{P}} = S_{\mathcal{P}}^{ch} \times S_{\mathcal{A}}^{spec}$ is the set of states, $s_0 = ((q_0, s_0^{ch}), s_0^{spec})$ is the initial state, $\delta_{\mathcal{P}} \subseteq \Delta_{\mathcal{P}}^{ch} \times \delta_{\mathcal{A}}^{spec}$ is the transition function, $\omega_{\mathcal{P}} : \delta_{\mathcal{P}} \to \mathbb{Z}_{\geq 1}$ is the weight function and $F_{\mathcal{P}} = S_{\mathcal{P}}^{ch} \times F_{\mathcal{A}}^{spec}$ is the set of final states. A transition $(p,s) \to (p',s) \in \Delta_{\mathcal{P}}$ if $p \to p' \in \Delta_{\mathcal{P}}^{ch}$ and $s \to^{\sigma} s'$. In the ME mode, $\sigma = \epsilon^{[d-1]} h_{\mathcal{P}}^{ch}(q'_c)$ if $c > 0$ and $\sigma = \epsilon^{[d]}$ otherwise, where c is the index of the active vehicle in p' and $d = \omega_{\mathcal{P}}^{ch}((p,s),(p',s'))$ is the duration of the transition. For the FC mode, $\sigma = \{h(q'_i) | i \in \{1,\ldots,N\}\} \in 2^{\Pi}$. As before, the weight function is inherited from the product $\mathcal{P}^{ch}$, i.e. for every $((p,s),(p',s')) \in \Delta_{\mathcal{P}}$ we have $\omega_{\mathcal{P}}((p,s),(p',s')) = \omega_{\mathcal{P}}^{ch}(p,p')$.

The algorithm to compute the product automaton $\mathcal{P}$ is outlined in Algorithm 26. A feasible and satisfying control policy is computed in Algorithm 27 as a projection on $\mathcal{T}^N$ of a path from $\mathcal{P}$. In Algorithm 27, $\beta_{\mathcal{T}}$ denotes the canonical projection on the $\mathcal{T}$-component of the product.

We now show that Algorithm 27 produces a feasible and satisfying control policy if one exists, thus solving Problem 6.1. The following statements are true for both

modes of operation (ME and FC). The only difference is in the construction of the product automaton $\mathcal{P}$, line 1 in Algorithm 27.

In the following, for two vectors b and b', $b' \geq b$ if the relationship holds component-wise. The following proposition holds trivially.

Proposition 6.1. *Let* $\mathbf{v}$ *be a feasible control policy starting with an initial charging state* b*. Then* $\mathbf{v}$ *is a feasible control policy starting with any initial battery state* $b' \geq b$*.*

Theorem 6.2. *Algorithm 27 is complete.*

Proof. Let q_0 be the initial state of the N vehicle system. First, we reduce the problem using Proposition 6.1, which implies that if a control policy $\mathbf{v}$ is feasible then we can construct another control policy from it by appending at the end of each loop a NFT such that every loop starts with the vehicles fully charged. Thus, in order to asses feasibility we only need to check reachability between states where all vehicles are docked and fully charged. Consider the graph $G = (V = P_N(\mathcal{C}), E)$ from Algorithm 27. The existence of a control policy is equivalent with the existence of an infinite path in G. This in turn implies that there must be a cycle in G reachable from the initial state q_0. If we assume that there is no such cycle, then all paths starting at q_0 are finite, which implies that no control policy exists. It follows that Algorithm 27 is complete. □

6.3.5 Optimality

In Section 6.3.4, we showed that if Problem 6.1 admits a solution, then there is a feasible control policy which has a prefix-suffix structure that can be computed on a finite graph. In this section, we will establish the same result for the optimal version of the problem (Problem 6.2) corresponding to the two cost functions J_1 and J_2.

Let $G^{opt} = (V, E, w)$ be a weighted graph, where $V = S_{\mathcal{P}}^{ch}$ is the vertex set. As in Algorithm 27, we proceed to construct the edge set E such that $(q, q') \in E$ if there

is a satisfying path in $\mathcal{P}$ starting at (q, s_0^{spec}) and ending at (q', s_f^{spec}), $s_f^{spec} \in F^{spec}$. The weight of $w(q, q')$ is equal to the minimum loop time. Note that loops that are not minimal can be replaced by the minimal ones to decrease the overall cost. To minimize J_1, a cycle in G^{opt} with minimum maximum weight must be computed, because the objective is to minimize the maximum loop time of any loop that is repeated infinitely often. The J_2 criterion is attained by a cycle in G^{opt} of minimum average weight as shown by Proposition 6.3.

Proposition 6.3. *Let $G = (V, E, w)$ be a strongly connected graph with possible self-loops and a weight map $w : E \to \mathbb{R}_+$. There is a path $\mathbf{v}^* = v_1, \ldots v_p\ (v_{p+1}, \ldots v_{p+s})^\omega$ that minimizes J_2 and $J_2(\mathbf{v}^*) = \frac{1}{s}(w(v_{p+s}, v_{p+1}) + \sum_{i=1}^{s-1} w(v_{p+i}, v_{p+i+1}))$.*

Proof. It is easy to see that ignoring any finite prefix of $\mathbf{v}$ does not change the value of J_2. Let $\mathbf{c}_s$ be the minimum average weight cycle in G and $\mathbf{v}$ be an infinite path. Since $\mathbf{v}$ is infinite and V is finite it follows that there is a node $v_{inf} \in V$ such that v_{inf} appears infinitely often in $\mathbf{v}$. Any finite sub-sequence of $\mathbf{v}$ delimited by v_{inf} defines a cycle. However, since $\mathbf{c}_s$ has minimum average weight it follows that each cycle in $\mathbf{v}$ delimited by v_{inf} has a greater cost that $\mathbf{v_s}$. This in turn implies that $J_2(v) \geq J_2((v_s)^\omega)$. □

Remark 6.4. *Finding the minimum average weight cycle is NP-complete. The reduction can be made from the Hamiltonian cycle problem. Therefore, we have to impose some additional restrictions on the control policies in order to reduce G^{opt} to a manageable size.*

6.3.6 Complexity

The complexities of Algorithm 26 are different for the ME and FC modes. The construction of the product transition system is $\mathcal{O}\left(\frac{|\mathcal{C}|!}{(N-|\mathcal{C}|)!} + N\,|R|\,\frac{|\mathcal{C}|!}{(N-|\mathcal{C}|)!} + N\,|\Delta|\right)$ for

the ME mode and $\mathcal{O}\left(((|Q|+|\Delta|)d_{max}t_{col})^N\right)$ for the FC mode, where d_{max} is the maximum duration of an edge in Δ. Constructing the charging FSA takes $\mathcal{O}\left(t_{ch}^N(N^2 d_{max})\right)$ and $\mathcal{O}\left(t_{ch}^N 2^N\right)$ in ME and FC modes, respectively. We describe the complexity of the next steps of Algorithm 26 in a unified manner. However, the actual complexity differs between modes for steps 3 and 6, because they depend on the size of the product transition system and the charging FSA. The complexity of these steps are as follows: $\mathcal{O}\left(\left|\tilde{Q}\right| \left|S_{\mathcal{A}}^{ch}\right| + \left|\tilde{\Delta}\right| \left|\delta_{\mathcal{A}}^{ch}\right|\right)$ for constructing the first product, $\mathcal{O}\left(\|\phi\| \, |\phi|\right)$ for converting the TWTL formula ϕ to scLTL formula ψ, where $|\phi|$ is the length of the formula (number of operators and propositions), $\mathcal{O}(2^{\mathcal{S}} 2^{2^{|\psi|}})$ for converting the scLTL formula to an FSA using *scheck* and $\mathcal{O}\left(\left|S_{\mathcal{P}}^{ch}\right| |S_{\mathcal{A}}^{spec}| + \left|\delta_{\mathcal{P}}^{ch}\right| |\delta_{\mathcal{A}}^{spec}|\right)$ for the final product automaton.

The complexity of Algorithm 27 is $\mathcal{O}\left(\frac{|\mathcal{C}|!}{(N-|\mathcal{C}|)!}(S_{\mathcal{P}} + \delta_{\mathcal{P}})\right)$ for constructing the graph $G = (V, E)$ and $\mathcal{O}(V + E)$ to test if there is a reachable cycle from the initial state. Obtaining an optimal solution is NP-complete and therefore exponential in $|V|$.

It is not surprising that the proposed algorithms have exponential complexity, because the VRP problem itself is NP-hard. However, the one outstanding question is how our approach compares to a MILP formulation in terms of scalability w.r.t. the number of vehicles N. The automata-based approach is well suited for the persistent VRP problem because it decreases the worst-case complexity over a MILP implementation. In our approach, we compute a product automaton once and from it we can compute control policies for loops. We then solve an NP-complete problem on the one-loop reachability graph, whose vertex set is polynomial in the number of robots N. Thus, the overall procedure has worst-case complexity $O(2^N + N^{k+1}2^N)$, where k is the fixed difference between the number of depos and robots. On the other hand, a MILP approach does not reuse previous computation and redundant

operations may be performed. As such, the worst-case complexity is $O(2^N + N^{2k}2^N)$. This analysis only considers N as a variable (the other parameters are fixed) and that the robots are identical. If we lift the latter assumption, the difference in complexity becomes even greater, because the size of the vertex set of the one-loop reachability graph becomes factorial in N. Thus, the automata-based approach has complexity $O(2^N + N(N+k)!2^N)$ and MILP has $O(2^N + (N+k)!^2 2^N)$. In practice, a MILP approach may be faster in computing a solution for a single loop, but since we need to perform the operation repeatedly the automata approach may be faster overall. Also, encoding the whole problem as a MILP program leads to 2-EXPTIME(N) complexity.

6.3.7 Generalizations

The presented framework can easily be modified to account for differences among vehicles, with respect to both motion and replenishing models. Different motion models can be specified as different individual transition systems $\{\mathcal{T}_i\}_1^N$, which can be used to construct the product transition system $\mathcal{T}^N$. The resource model can also be customized with vehicle specific charging and operation times, each satisfying the assumptions from Section 6.3.2. The framework can also be minimally modified to support the case when $\frac{t_{op}}{t_{ch}} \in \mathbb{Z}_{\geq 1}$, e.g., when the resource is fuel. In this case, t_{ch} and t_{op} are interchanged in the construction of the state set of the charging FSA $\mathcal{A}^{ch}$ and the inverse of γ is used in the update rule.

Also, the proposed algorithms can be used for the case when the weights of the edges in Δ are upper bounds for travel times, rather than fixed durations. In this case, worst-case feasible control policies are computed off-line, i.e. as described above, and replanning is performed on-line when the actual transition durations become available.

6.4 Implementation, Results, and Experimental Validation

We implemented the algorithms developed in this paper in a software tool that takes as input an environment topology (i.e., the positions of the sites and charging stations), the durations of the motions, the operation, charging times, and collision times, and a mission specification in the form $\mathbb{G}\phi$, where ϕ is a TWTL formula. The output is a vehicle control policy. The tool, which was implemented in Python2.7, uses the LOMAP (Ulusoy et al., 2013c) and networkx (Hagberg et al., 2008) packages to manipulate and process automata. The tool has an input-output graphical user interface (Figure 6·1 was generated using the tool).

The tool, running on a Linux system with a 2.1 GHz processor and 32GB memory, was used to generate control policies for the case study presented in Examples 6.1 and 6.2. The TWTL formula ϕ_{tw} was translated to an FSA with 1468 states and 5845 transitions. In the ME mode, the construction of $\mathcal{T}^N$, $\mathcal{A}^{ch}$, $\mathcal{P}^{ch}$, and $\mathcal{P}$ took 1.7 msec, 491 msec, 13.5 sec, and 16 sec to compute. Their sizes were 24, 3721, 89304, 75538 states and 108, 127734, 332328, 263144 transitions, respectively. The test for feasibility took 11 sec to execute. Note that the size of the final product automaton $\mathcal{P}$ is not larger than the size of $\mathcal{P}^{ch}$. This is due to the implementation of the construction of $\mathcal{P}$, which contains only the states reachable from the initial ones.

A ME control policy is given in (6.2) from Example 6.1. The control policy is feasible and satisfies the specification. Furthermore, it is also optimal with respect to both J_1 and J_2 with the assumption that vehicles start each loop fully charged. Without this assumption, the optimization problem would have to be solved on a graph with 89304 vertices, which is intractable (see Section 6.3.5).

Using the FC mode and the same setup, but with $t_{op} = 20$ and $t_{ch} = 40$, the construction of $\mathcal{T}^N$, $\mathcal{A}^{ch}$, $\mathcal{P}^{ch}$, and $\mathcal{P}$ took 662 msec, 387 msec, 25.8 min and 35.2 min

to compute. Their sizes were 3066, 1681, 5153946, 6487656 states and 5200, 24964, 7942452, 9669808 transitions, respectively. Feasibility was established in 6.65 min. A feasible and satisfying control policy for the FC mode is

$$\begin{aligned} v^{FC}(1) &= \left(Ch_3^{[2]}\vec{Ch}_3^{[3]}C^{[4]}\vec{A}^{[2]}A^{[3]}\vec{Ch}_2^{[6]}Ch_2^{[40]}Ch_2^{[1]}\vec{Ch}_2^{[3]}B^{[2]}\vec{Ch}_3^{[3]}Ch_3^{[49]}\right)^{\omega} \\ v^{FC}(2) &= \left(Ch_2^{[1]}\vec{Ch}_2^{[3]}B^{[2]}\vec{Ch}_3^{[3]}Ch_3^{[49]}Ch_3^{[2]}\vec{Ch}_3^{[3]}C^{[4]}\vec{A}^{[2]}A^{[3]}\vec{Ch}_2^{[6]}Ch_2^{[40]}\right)^{\omega} \end{aligned}$$

and the corresponding output word is

$$o = \left(\epsilon^{[3]}B^{[2]}\{B,C\}^{[1]}C^{[3]}\epsilon^{[2]}A^{[3]}\epsilon^{[46]}\right)^{\omega},$$

where $\{B,C\}^{[1]}$ indicates that both sites B and C are occupied at the same time.

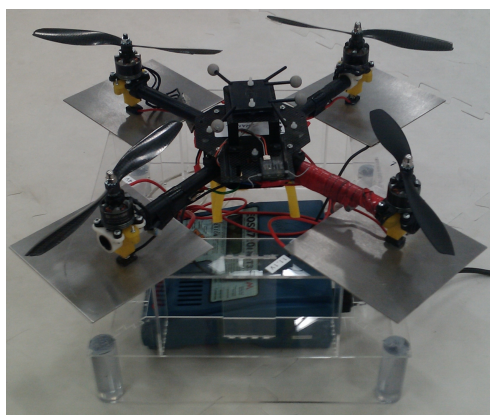


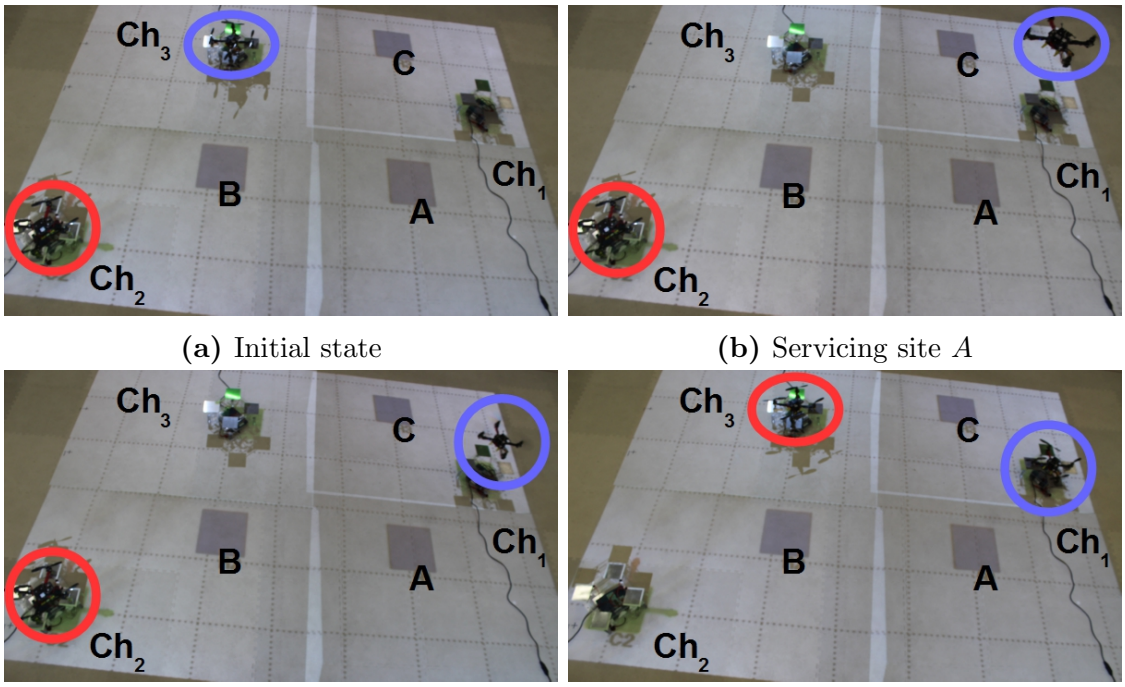
Figure 6·2: Quadrotor docked at a charging station.

The above case study was also implemented in our aerial vehicle experimental setup, which consists of a team of quadrotors flying autonomously in an indoor space equipped with a motion capture system, short-throw projectors that generate images on the floor, and fully automatic charging stations that can detect the presence of a vehicle and its charging level (see Figures 6·2 and 6·3). To generate transitions among sites and charging stations, the 3D space was partitioned into small rectangular regions. Using the framework developed in (Belta and Habets, 2006), vector fields were

designed in each rectangle to guarantee safe transitions between adjacent rectangles and stabilization to the center of a rectangle (i.e., for servicing a site, which corresponds to hover). The tool developed in (Gol and Belta, 2013) was used to determine the (upper bounds for the) durations of the transitions. The durations of the landing and take-off maneuvers at the charging stations were included in the durations of the transitions to and from the charging stations. Quadrotor feedback control laws were generated to follow the designed vector fields by using the framework developed in (Zhou and Schwager, 2014).

We used the same setup as described in Example 6.1 (Figure 6·1) with the specification from Example 6.2. We consider the MC mode and a time interval Δt of 6 sec. Four snapshots from a successful experimental trial are shown in Figure 6·3. Consider the loop starting at Ch_3 and Ch_2 and ending at Ch_1 and Ch_3, respectively.

In this loop, the blue quadrotor visits site C, services C for at least 18 sec, visits site A, services A for at least 12 sec, and lands at Ch_1. After the first blue quadrotor lands, the red one takes-off and visits site B, services B for at least 12 sec, visits site C, services C for 12 sec, and lands at Ch_3. The actual transition and servicing durations were 21.47, 18.0, 5.55, 12.0, 23.9, 23.08, 12.0, 28.19, 12.01, 13.72 (all in sec and in the order described above). These durations are bounded above by the estimated durations that were used to compute the control policy, which were 24, 18, 18, 12, 24, 24, 12, 30, 12, 30. Note that specification ϕ_{tw} is satisfied, because: (1) A is serviced in 57.01sec, before its deadline of 72 sec; (2) C is serviced in 38.47 sec, before its deadline of 54 sec; (3) B followed by C are serviced in 156.2 sec, before the deadline of 192 sec; and (4) C is serviced in 42.2 sec after B, before the deadline of 48 sec.



(a) Initial state **(b)** Servicing site A

(c) Docking at Ch_1 **(d)** State at the end of the loop

Figure 6·3: Two quadrotors in an environment with three sites and three charging stations. Figure 6·3a: the quadrotors are fully charged and docked at the start of the mission. Figure 6·3b: the blue quadrotor is servicing site A, while the red quadrotor is still docked at charging station Ch_2. The docking procedure is shown in Figure 6·3c. The blue quadrobot attempts to land on charging station Ch_1. At the end of the first loop (Figure 6·3d), the quadrotors are docked at Ch_1 and Ch_3.

Chapter 7

Dynamic Persistent Vehicle Routing Problem with Charging and Temporal Logic Constraints

This chapter addresses a persistent vehicle routing problem, where a team of vehicles is required to achieve a task repetitively. The task is given as a Time-Window Temporal Logic (TWTL) formula defined over the environment. The fuel consumption of each vehicle is explicitly captured as a stochastic model. As vehicles leave the mission area for refueling, the number of vehicles may not always be sufficient to achieve the task. We propose a decoupled and efficient control policy to achieve the task or its minimal relaxation. We quantify the temporal relaxation of a TWTL formula and present an algorithm to minimize it. The proposed policy has two layers: 1) each vehicle decides when to refuel based on its remaining fuel, 2) a central authority plans the joint trajectories of the available vehicles to achieve a minimally relaxed task. We demonstrate the proposed approach via simulations and experiments involving a team of quadrotors that conduct persistent surveillance. The framework presented in this chapter differs from the one in Chapter 6 in four aspects: (a) it is developed for on-line execution, (b) it assumes stochastic fuel models, (c) it penalized collisions between vehicles instead of requiring avoidance, and (d) it considers relaxed specifications.

7.1 Problem Formulation

In this section, we formulate a VRP based on a persistent surveillance scenario. **Notation:** $q^{[d]}$ denotes d repetitions of q.

7.1.1 Environment Model

Consider an environment that contains a set of monitoring sites ($\mathcal{S}$) and a set of bases (or charging stations) ($\mathcal{C}$). Let $\mathcal{E} = (Q, \Delta, \varpi)$ denote a weighted directed connected graph, where $Q = \mathcal{S} \cup \mathcal{C}$ is the set of nodes representing the sites and the bases, $\Delta \subseteq Q \times Q$ is the set of edges representing the feasible travel between the nodes, and $\varpi : \Delta \to \mathbb{Z}_{\geq 1}$ is the edge weight that represents the travel time between the nodes. In this setting, we assume that there exists a path from any site to one of the bases without visiting any other sites (e.g., dashed edges in Figure 7·1a).

7.1.2 Vehicle Model

Given $\mathcal{E} = (Q, \Delta, \varpi)$, a team of vehicles move on the edges Δ to pursue persistent operations. For any $q \in Q$, $\vec{q}$ denotes moving towards q. Let $\vec{Q} = \{\vec{q} | q \in Q\}$. At any t, the state of vehicle i is $[f_i(t), x_i(t)]$, where $f_i(t)$ is its remaining fuel, and $x_i(t) \in Q \cup \vec{Q}$ is its target state (i.e., either the node it is occupying or the node it is traveling to). In this paper, we only focus on the high-level planning. We assume that low-level controllers drive the vehicles from their current states to the designated target states (more information is provided in Section 7.4.2 for a case when the vehicles are quadrotors).

Communication Model: We assume that 1) each vehicle can communicate with all the other vehicles through a complete communication graph, 2) there is no cost in communication, and 3) the information propagates significantly faster than the motion of the vehicles.

Fuel Model: Each vehicle has limited fuel capacity and consumes fuel unless it is located at a base. Accordingly, we use the following stochastic fuel model:

$$f(t+1) = \begin{cases} \min\left\{f(t) + \delta f^c, f_{max}\right\} & \text{if } at\ base, \\ f(t) - \delta f^d + \xi(t) - \delta f^p & \text{otherwise,} \end{cases} \tag{7.1}$$

where $\delta f^c > 0$ is a constant refuel rate at the base, $\delta f^d > 0$ is a constant fuel consumption while operating, f_{max} is the maximum fuel capacity, $\xi(t)$ is a random variable modeling uncertainty in the fuel consumption, and $\delta f^p \in \{0, \beta_1, \beta_2\}$ models a fuel penalty if the vehicles avoid collisions through some maneuvers. In other words, if multiple vehicles travel the same edge or operate at the same node, they avoid each other by modifying their trajectories, e.g., a change in flight altitude. Such operational changes typically cause more fuel consumption. Thus, $\delta f^p = \beta_1 > 0$ for each vehicle traveling the same edge or occupying the same node; $\delta f^p = \beta_2 > \beta_1$ for each vehicle traveling the same edge and arriving the same node simultaneously; $\delta f^p = 0$ in other cases.

7.1.3 Control Policy

In a persistent surveillance mission, each vehicle needs to (i) avoid running out of fuel, and (ii) work collaboratively to achieve a desired objective. Thus, each vehicle needs an efficient decision for when to refuel and how to move. In this paper, we propose to decouple the decision-making for refueling and operating in the surveillance area. In the proposed policy, each vehicle has a label as *active* or *inactive*. A vehicle changes its label from *active* to *inactive* if it decides to return to the base, whereas its label switches from *inactive* to *active* when it arrives at the surveillance area after refueling. We assume that each vehicle broadcasts any change in its label through the communication network. Then, a central authority assigns a target node to each active vehicle. Consequently, each vehicle's trajectory depends on two policies: the

refuel policy results in a strategic decision for safe return to a base, and the *operational control policy* results in efficient movement in the surveillance area.

Refuel Policy

In this paper, the vehicles follow a threshold policy for refueling. Accordingly, given a fuel threshold $f_i^{cr}(t)$ for an active vehicle i, if $f_i(t) > f_i^{cr}(t)$, then i remains to be active and it is in the surveillance area. Otherwise, i is inactive and moves towards a base.

Operational Control Policy

For M active vehicles, the operational control policy is a sequence $\mathbf{\Pi}_M = \pi(t)\pi(t+1)\ldots$ where $\pi(t) \in (Q \cup \vec{Q})^M$ specifies at each time t and for each vehicle $i \in \{1, \ldots, M\}$ where to stay or to go at $t+1$. We denote $\pi_i(t)$ as the target state of i at t and π_i as the control policy for i (the sequence of the target states).

7.1.4 Problem Definition

In this paper, achieving a persistent task means infinitely many satisfactions of a TWTL formula ϕ (i.e., $\mathbb{G}\phi$ where $\mathbb{G}$ stands for *always*). To formalize this concept, we define the infinite concatenation closure of ϕ as the concatenation of infinitely many copies of ϕ, i.e., $(\phi \cdot \phi \cdot \ldots)$. Similarly, we define the infinite concatenation closure of relaxed TWTL formulae as $(\phi(\boldsymbol{\tau}^1) \cdot \phi(\boldsymbol{\tau}^2) \cdot \ldots)$, where any $\phi(\boldsymbol{\tau}^i)$ corresponds to a $\boldsymbol{\tau}^i$-relaxation of ϕ. Note that a control policy $\mathbf{\Pi}_M = \pi(1)\pi(2)\ldots$ induces an output word $\mathbf{o}$.

Definition 7.1 (Output word)**.** *The output word generated by a control policy,* $\mathbf{\Pi}_M = \pi(1)\pi(2)\ldots$*, is* $\mathbf{o} = o_1 o_2 \ldots$*, where* $o_t = \{\pi_i(t) | \pi_i(t) \in \mathcal{S}, i \in \{1, \ldots, M\}\}$ *is the set of all monitoring sites occupied by* M *vehicles at time* t*.*

Ideally, it is desired to find a policy that generates $\mathbf{o}$ satisfying $(\phi(\boldsymbol{\tau}^1)\cdot\phi(\boldsymbol{\tau}^2)\cdot\dots)$, where $\boldsymbol{\tau}^1 = \boldsymbol{\tau}^2 = \cdots = \mathbf{0}$. However, $\boldsymbol{\tau}^i$ may contain nonzero elements due to uncertain vehicle availability in the surveillance area. In that case, the objective becomes to find a policy that minimizes $|\boldsymbol{\tau}^i|_{TR}$, i.e., the temporal relaxation.

Problem 7.1. *Given an environment $\mathcal{E} = (\mathcal{S} \cup \mathcal{C}, \Delta, \varpi)$, M active vehicles, and a persistent task $\mathbb{G}\phi$, let Π_M generate an output word $\mathbf{o}$ that satisfies $(\phi(\boldsymbol{\tau}^1)\cdot\phi(\boldsymbol{\tau}^2)\cdot\dots)$. Find an optimal operational control policy*

$$\Pi_M^* = \arg\min_{\Pi_M} |\boldsymbol{\tau}^i|_{TR}\ , \quad \forall i. \tag{7.2}$$

Note that if M is constant during the mission, Π_M^* results in the optimal trajectories minimizing the temporal relaxation. However, M varies during the mission due to fuel uncertainty. Thus, solving (7.2) as M changes results in switching control policies. In Sec. 7.3, we show that there always exists a solution under the switching policies and our proposed algorithm can find one. Nonetheless, resulting trajectories under the switching policies are not necessarily optimal.

7.2 Control Synthesis

Our proposed solution to Problem 7.1 (Algorithm 28) is inspired from automata-based model checking and has two phases. In the *off-line* computations, first, a list of active modes are created (e.g, a total number of N vehicles corresponds to N modes, where mode n represents the presence of n active vehicles in the environment). For each mode, a transition system is generated from the environment model. These are then combined with a special finite state automaton to obtain a list of product automata, each of which captures both motion and satisfaction in the corresponding mode. In the *on-line* computations, a centralized controller uses the product automata

to compute the target states of all active vehicles. To this end, the control policy, Π_M, is computed on the currently active product automaton using a Dijkstra-based algorithm, and it is recomputed if any change occurs in M. Overall, we propose a hybrid control policy shown in Algorithm 28.

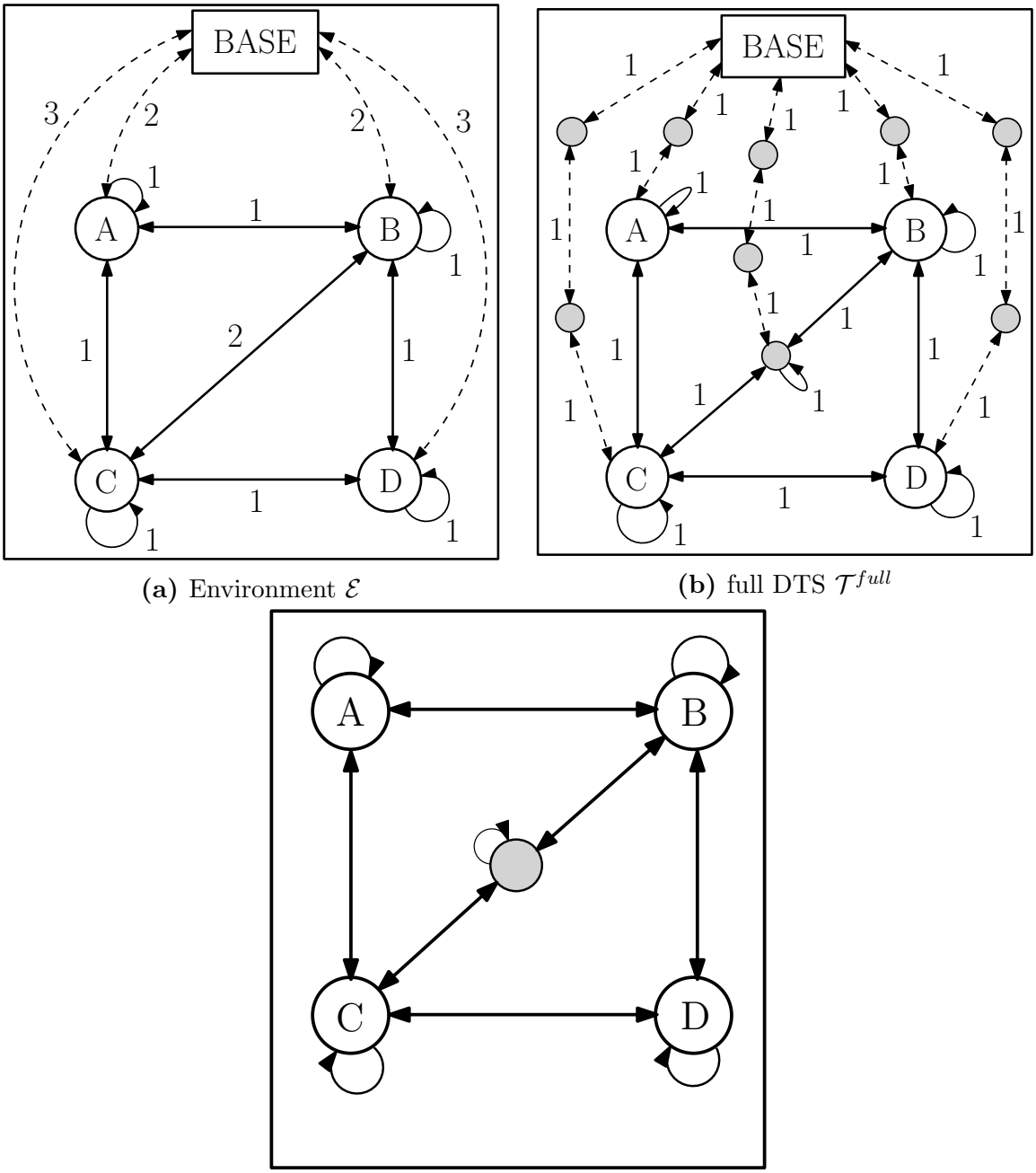
Algorithm 28: Hybrid Control Policy

```
Input: ℰ = (Q, Δ, ϖ) environment, ϕ TWTL formula, N number of vehicles
Extract 𝒯 and 𝒯^full from ℰ and construct 𝒯^k, 1 ≤ k ≤ N   // Off-line
𝒜_∞ ← translate(ϕ), the FSA corresponding to ϕ(∞)
Create product automata 𝒫^k = 𝒯^k × 𝒜_∞, 1 ≤ k ≤ N
while True do                                                // On-line
    foreach active vehicle do
        if vehicle.fuel is critical then
            vehicle.mode ← inactive;  controls.inactive ← returnToBase()
    controls.active ← operationalPolicy(active vehicles)
    foreach vehicle do
        vehicle.move();   vehicle.updateFuel()
        if vehicle.state ∈ Q̂ (surveillance area) then vehicle.mode ← active
        else vehicle.mode ← inactive
```

7.2.1 Multiple-Vehicle Motion

The motion model of a single vehicle is captured by a *deterministic transition system*, $\mathcal{T} = (Q, q_0, \Delta, AP, h)$, where AP is the set of observations.

The DTS of a vehicle is obtained by transforming the environment graph into an unweighted directed graph. To this end, we split up all transitions to have an edge weight of 1 and define new auxiliary states on the divided edges. Let $\mathcal{S}^{aux}$ and $\mathcal{C}^{aux}$ denote the set of auxiliary states between the sites and between the sites and the bases, respectively. The DTS of a vehicle is $\mathcal{T}^{full} = (Q^{full}, q_0, \Delta^{full}, AP, h)$, where $Q^{full} = \mathcal{S} \cup \mathcal{C} \cup Q^{aux}$ and $Q^{aux} = \mathcal{S}^{aux} \cup \mathcal{C}^{aux}$; $q_0 \in Q^{full}$; $\Delta^{full} \subseteq Q^{full} \times Q^{full}$; $AP = \mathcal{S} \cup \{\epsilon\}$ where ϵ indicates that no site is occupied; and $h : Q^{full} \to AP$ is the



(a) Environment $\mathcal{E}$

(b) full DTS $\mathcal{T}^{full}$

(c) reduced DTS $\mathcal{T}$

Figure 7·1: (a) Environment $\mathcal{E}$ containing four monitoring sites A, B, C, D and a base, (b) the full motion DTS on $\mathcal{E}$, and (c) the reduced DTS modeling motion only in the surveillance area.

labeling function such that it assigns a label to each site, i.e., $h(q) = q$ if $q \in \mathcal{S}$ and $h(q) = \epsilon$ for $q \notin \mathcal{S}$.

For example, Figure 7·1b illustrates the *full DTS* of a vehicle, which is extracted from the environment in Figure 7·1a. We also define a *reduced DTS* that disregards the bases as $\mathcal{T} = (Q^{red}, q_0, \Delta^{red}, AP, h)$, where $Q^{red} = \mathcal{S} \cup \mathcal{S}^{aux}$ involves only the sites and the auxiliary states connecting them as shown in Figure 7·1c.

We use $\mathcal{T}$ for the planning of active vehicles in the surveillance area. This enables to decouple the operational planning from the refuel decisions. Moreover, using a reduced DTS in planning significantly reduces the state-space of the overall system since the concurrent motion of the vehicles is represented by a *product transition system.*

Definition 7.2. *A* Product Transition System (PTS) $\mathcal{T}^k$ *for* $k \geq 1$ *is a DTS* $\mathcal{T}^k = (Q^k, q_0^k, \Delta^k, 2^{AP}, h^k)$*, where* $Q^k \subseteq Q^{red} \times \cdots \times Q^{red}$ *for* k *times is the set of states;* $q_0^k \in Q^k$ *is the initial state;* $\Delta^k \subseteq Q^k \times Q^k$ *is the set of transitions such that* $([x_1, \ldots x_k], [x'_1, \ldots x'_k]) \in \Delta^k$ *if* $(x_i, x'_i) \in \Delta^{red}$ *for all* $i \in \{1, \ldots, k\}$*;* 2^{AP} *is the set of observations (power set of AP); and* $h^k([x_1, \ldots, x_k]) = \{h(x_i) | i \in \{1, \ldots, k\}\}$.

In the proposed approach, the centralized controller only tracks the occupied states of $\mathcal{T}$ with multiplicities. Thus, we use quotient PTSs, whose states are equivalence classes induced by the permutation of the state vectors (e.g., the states (A, A, B), (A, B, A) or (B, A, A), representing 3 vehicles occupying A and B, are merged into a single state). This representation greatly reduces the sizes of the resulting PTSs, and any PTS along the paper implies a quotient PTS.

7.2.2 Specification

The specification is enforced using a *deterministic finite state automaton.* Note that $\mathcal{A}$ can be constructed from any ϕ. However, $\mathcal{A}$ represents only the specification with

the given time windows. In order to compactly represent all temporal relaxations of ϕ, a special automaton $\mathcal{A}_\infty$ is constructed based on the procedure in Chapter 5. Accordingly, $\mathcal{A}_\infty$ represents $\phi(\boldsymbol{\tau})$ for all possible $\boldsymbol{\tau}$.

7.2.3 Operational Control Policy

The operational control policy is computed on the product automata between the PTSs and $\mathcal{A}_\infty$, which capture both the motion of the active vehicles and satisfaction of the formula.

Definition 7.3. *A* Product Automaton (PA) $\mathcal{P}_k = \mathcal{T}^k \times \mathcal{A}_\infty$ *for* $1 \leq k \leq N$ *is a tuple* $\mathcal{P}_k = (S_{\mathcal{P}_k}, (q_0^k, s_0), \Delta_{\mathcal{P}_k}, F_{\mathcal{P}_k})$*, where* $S_{\mathcal{P}_k} = Q^k \times S_{\mathcal{A}_\infty}$ *is the finite set of states;* $(q_0^k, s_0) \in S_{\mathcal{P}_k}$ *is the initial state;* $\Delta_{\mathcal{P}_k} \subseteq S_{\mathcal{P}_k} \times S_{\mathcal{P}_k}$ *is the set of transitions;* $F_{\mathcal{P}_k} = Q^k \times F_{\mathcal{A}_\infty}$ *is the set of accepting states.*

A transition $\big((q,s),(q',s')\big) \in \Delta_{\mathcal{P}_k}$ implies $(q,q') \in Q^k$ and $s \xrightarrow{h(q)}_{\mathcal{A}_\infty} s'$. The notions of trajectory and acceptance are the same as in FSA. A satisfying run of $\mathcal{T}^k$ with respect to ϕ can be obtained by computing a path from the initial state to an accepting state over $\mathcal{P}_k$ and projecting the path onto $\mathcal{T}^k$.

We propose Algorithm 29 (line 8 in Algorithm 28) to compute the target states of the active vehicles at each time step. Algorithm 29 stores a local policy generated on the currently selected PA $\mathcal{P}$ and the last returned PA state $p = (q,s) \in S_{\mathcal{P}}$, where q is the PTS state, and s is the state on $\mathcal{A}_\infty$. The switching between PAs occurs when the last stored q is different than the actual PTS state of active vehicles q' (line 4). When s reaches an accepting state and the policy becomes empty, s is set to the initial state of $\mathcal{A}_\infty$ and the next satisfaction of ϕ initiates (line 8). Using $\mathcal{P}$ and p, the target states of the active vehicles are computed in line 10 by *computePolicy*(), which proceeds by traversing the structure of ϕ from smaller to larger sub-formulae. It uses special annotation on the automaton $\mathcal{A}_\infty$ to compute satisfying paths in $\mathcal{P}$

without considering within operators. Then, these paths are recursively filtered and extended based on the boolean and temporal operators connecting them. If there is no satisfying policy, then the procedure returns the current p. The detailed description of $computePolicy()$ can be found in Chapter 5. The target states are distributed to vehicles via Hopcroft-Karp algorithm (line 11).

Algorithm 29: On-line planning – $operationalPolicy()$

Input: the set of active vehicles
Output: the next state for each active vehicle
Data: $p = (q, s)$ – last PA state,$\mathcal{P}$ – selected PA, $policy$ – current policy

1 **if** *no active vehicles* **then return** $\{\}$
2 $q' = [vehicle.state \mid vehicle.mode = active]$
3 $replan = False$
4 **if** $q \neq q'$ **then** `// any change in the PTS state`
5 $\quad \mathcal{P} \leftarrow \mathcal{P}_{|q'|}; \quad q \leftarrow q'; \quad replan \leftarrow True$ `// switch PA`
6 **else**
7 $\quad$ **if** $policy \neq \{\}$ **then** $p = policy.next()$ **else** $replan \leftarrow True$
8 **if** $s \in F_{\mathcal{A}_\infty}$ **then** $s \leftarrow s_0$ `// update FSA state`
9 **if** $replan = True$ **then**
10 $\quad policy \leftarrow computePolicy(\mathcal{P}, p); \quad p \leftarrow policy.next()$
11 **return** $distributeControls(q)$

7.3 Analysis of the Hybrid Control Policy

In this section, we discuss the performance, safety, and complexity of the proposed control policy.

7.3.1 Performance

First, we show that a relaxed TWTL formula can always be satisfied under an assumption on vehicle capabilities.

Definition 7.4 (Operational Cycle)**.** *An operational cycle of a vehicle is an ordered*

sequence of traveling to the area, operating in the area, returning to the base, and refueling.

Definition 7.5 (Formula Primitive)**.** *Given ϕ, a formula primitive is a maximal subtree in $AST(\phi)$, which does not contain a within operator.*

Assumption 2. *Given ϕ, a vehicle is able to satisfy any ϕ primitive(s) at least once in one operational cycle.*

Consider $\phi = [H^4 A]^{[3,8]} \wedge [H^2 B \cdot H^1 C]^{[4,9]}$ whose formula primitives are $H^4 A$ and $H^2 B \cdot H^1 C$. Assumption 2 implies that, in one operational cycle, the vehicle can reach A, stay there for 4 time steps, and return to the base safely. Similarly, it can also reach B, stay there for 2 time steps, then reach C, stay there for 1 time step, and return to the base.

Definition 7.6 (Feasible Sequence of Formula Primitives)**.** *Given ϕ, a feasible sequence of formula primitives is an ordered sequence of formula primitives, whose overall satisfaction implies a feasible relaxation of ϕ.*

Again, consider $\phi = [H^4 A]^{[3,8]} \wedge [H^2 B \cdot H^1 C]^{[4,9]}$. There exist only two feasible sequences of formula primitives, which are $(H^4 A, H^2 B \cdot H^1 C)$ and $(H^2 B \cdot H^1 C, H^4 A)$.

Theorem 7.1. *Let ϕ be a TWTL formula. If Assumption 2 holds, then there always exists a* feasible sequence *that induces a valid relaxation $\phi(\boldsymbol{\tau})$ such that $\|\phi(\boldsymbol{\tau}^i)\| \leq k\, t^*_{OC}$, where k is the length of the longest feasible sequence of ϕ primitives, and t^*_{OC} is the maximum duration to finish a cycle.*

Proof. Let $\mathbb{T}$ be the AST obtained from $AST(\phi)$ by contracting each *primitive* to a single leaf node. Each intermediate node of $\mathbb{T}$ corresponds to either: (i) a *within* operator, or (ii) a binary operation $(\wedge, \vee, \cdot)$ and at least one child which corresponds to a *within operator*. It follows by structural induction that there exists a *feasible*

sequence of ϕ primitives $\boldsymbol{\vartheta} = \vartheta_1, ..., \vartheta_k,\ k \geq 1$, because either: (i) a sub-formula corresponding to the node can be relaxed, or (ii) a sub-formula associated with a child node is satisfied and the other child node is relaxed until the vehicles satisfy it. Let t^*_{OC} denote the maximum duration of an operational cycle among all vehicles. Based on Assumption 2, each ϑ_i can be satisfied one by one within t^*_{OC}. Thus, the relaxation induced by $\boldsymbol{\vartheta}$ has a finite bound given by $\|\phi(\boldsymbol{\tau})\| \leq k\, t^*_{OC}$. □

7.3.2 Safety

In Algorithm 28, the decision to refuel (be inactive) based on the threshold policy should ensure safe return to the base.

Proposition 7.2. *Let $\mathcal{N}_i(t)$ be the set of adjacent nodes to vehicle i on $\mathcal{T}^{full}$ and let $\bar{\xi}$ be the maximum uncertainty in the fuel consumption. Executing Algorithm 28 ensures safe return to the base for vehicle i, if the refuel policy has a threshold*

$$f_i^{cr}(t) \geq \max_{q_j \in \mathcal{N}_i(t)} (1 + \varpi_{q_j q_B})(\delta f^d + \bar{\xi} + 2\beta) \tag{7.3}$$

Proof. We will show that if (7.3) is satisfied, then vehicle i never runs out of fuel before reaching the base. According to the refuel policy, a vehicle is active, if it is operating in the surveillance area and $f_i(t) > f_i^{cr}(t)$. Thus, (7.3) implies $f_i(t) > \max_{q_j \in \mathcal{N}_i(t)} (1+\varpi_{q_j q_B})(\delta f^d+\bar{\xi}+2\beta)$. Based on (7.1), $f_i(t) > f_i(t+1) \geq f_i(t)-(\delta f^d+\bar{\xi}+2\beta)$. Using the previous inequalities, we get $f_i(t+1) > \max_{q_j \in \mathcal{N}_i(t)} \varpi_{q_j q_B}(\delta f^d + \bar{\xi} + 2\beta)$, where $q_j \in \mathcal{N}_i(t)$ is a state vehicle i can reach at $t+1$. Since $\max_{q_j \in \mathcal{N}_i(t)} \varpi_{q_j q_B}(\delta f^d + \bar{\xi} + 2\beta) \geq \varpi_{q_j q_B}(\delta f^d + \bar{\xi} + 2\beta)$, the vehicle has sufficient fuel to go back to the base from any $q_j \in \mathcal{N}_i(t)$ when its label is active at t. □

7.3.3 Complexity

In Algorithm 28, the construction of $\mathcal{T}$ and $\mathcal{T}^{full}$ has complexity $O(\sum_e \varpi_e)$, where ϖ_e is the weight of edge e in $\mathcal{E}$. For N vehicles, the complexity of constructing all PTS is $O\left(\binom{|Q^{red}|+N}{N}\right)$ since the size of PTS $\mathcal{T}^k$ is equal to the number of permutations of k objects from Q^{red} with repetitions, i.e., $\binom{|Q^{red}|+k-1}{k}$ (Vilenkin, 1971). Constructing $\mathcal{A}_\infty$ from ϕ has complexity $O(2^{|\phi|})$ where $|\phi|$ is the length of the formula (Vasile et al., 2016). Finally, the complexity of computing each PA $\mathcal{P}_k$ is $O(\left|Q^k\right| \cdot |S_{\mathcal{A}_\infty}|)$. In Algorithm 29, the on-line planning is $O(|S_{\mathcal{P}_k}| + |\Delta_{\mathcal{P}_k}|)$ for $1 \leq k \leq N$ (Vasile et al., 2016). Moreover, how to return to the base is computed by running Dijkstra's algorithm on $\mathcal{T}^{full}$, which gives a complexity of $O(\left|Q^{full}\right| + \left|\Delta^{full}\right|)$. Overall, we improve the complexity of the solution (compared to the one in (Vasile and Belta, 2014b)) by 1) using multiple smaller PAs instead of a single complex PA, 2) decoupling the refuel decision from the trajectory planning, which significantly reduces the state-space, 3) representing ϕ via the special automaton $\mathcal{A}_\infty$ in a more compact way, and 4) using quotient PTSs instead of the normal ones.

7.4 Case Study

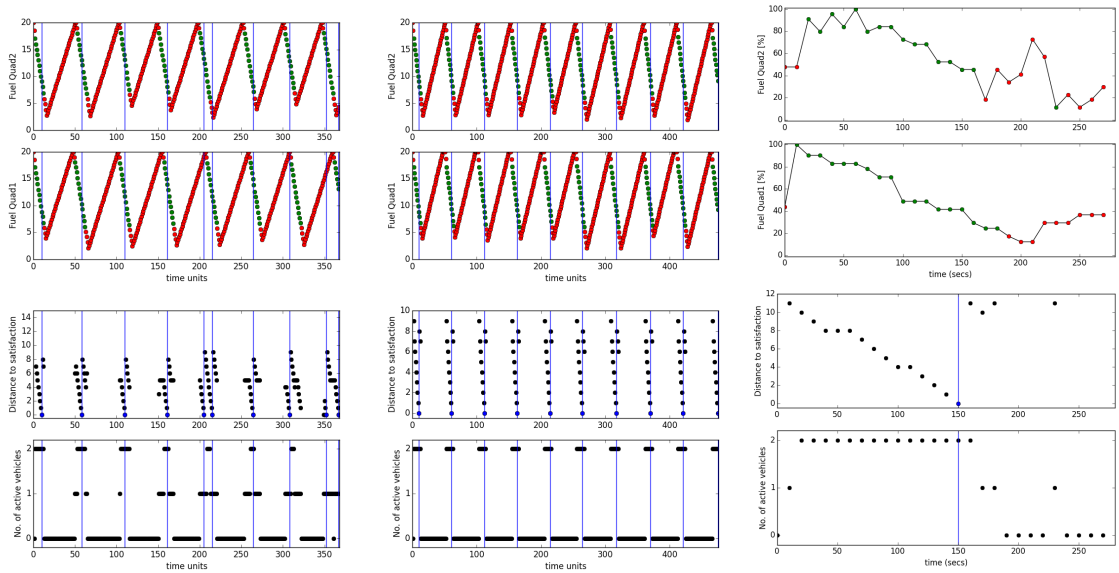
In this section, we show some simulations and experimental results for two identical quadrotors.

7.4.1 Simulation Results

We consider two identical vehicles, an environment with four sites and a base as in Figure 7·1a, and the TWTL formula $\phi = [H^2 A]^{[0,8]} \cdot [H^3 B \wedge [H^2 C]^{[1,5]}]^{[0,7]} \cdot [H^1 D]^{[0,3]}$, which means "perform in order: 1) service A for 2 time units within $[0, 8]$; 2) within $[0, 7]$, service B for 3 time units and service C for 2 time units within $[1, 5]$; and 3) service D for 1 time unit within $[0, 3]$". Note that $\|\phi\| = 20$ so a single satisfaction of

ϕ needs to be achieved in 20 time units. The parameters of each vehicle are selected as $f_{max} = 20$, $\delta f^c = 0.5$, $\delta f^d = 1$, $\delta f^p \in \{0, 0.2, 0.4\}$, $\xi(t) \sim \text{unif}(-0.1, 0.1)$.

The simulations were implemented in Python2.7 on a Intel Core i7 laptop with a 1.8 GHz processor and 8GB memory. ϕ was translated to $\mathcal{A}_\infty$ with 16 states and 36 transitions in 9 msec. The construction of $\mathcal{T}^1$, $\mathcal{T}^2$, $\mathcal{P}_1$, and $\mathcal{P}_2$ took <1 msec, 1 msec, 3 msec, and 19 msec, respectively. Moreover, $\mathcal{T}^1$, $\mathcal{T}^2$, $\mathcal{P}_1$, and $\mathcal{P}_2$ have 5, 15, 80, 240 states and 11, 78, 255, 2115 transitions, respectively. Overall, the off-line computation of Algorithm 28 took 65 msec, while a single iteration in its on-line computation took less than 1 msec.



(a) (b) (c)

Figure 7·2: (a) Simulation results: 10 satisfactions of the relaxed formulas via the proposed policy, (b) Simulation results: 10 satisfactions of the original formula via the benchmark policy, (c) Experimental results with two quadrotors.

The remaining fuel of each vehicle at each time step is displayed in Figure 7·2(a). The green and red markers indicate that the vehicle is *active* and *inactive*, respectively. In order to measure the progress towards satisfaction, we define the *distance to*

satisfaction (d_{sat}) at each t as the length of the shortest path in $\mathcal{A}_\infty$ from the current specification state to a final state. The vertical lines in this figure indicate a single satisfaction of $\phi(\boldsymbol{\tau}^i)$, which we call a satisfaction loop. The first 2 relaxed formulae satisfied by the vehicles are: $\phi(\boldsymbol{\tau}^1) = [H^2A]^{[0,8-\mathbf{4}]} \cdot [H^3B \wedge [H^2C]^{[1,5-\mathbf{2}]}]^{[0,7-\mathbf{4}]} \cdot [H^1D]^{[0,3-\mathbf{2}]}$; $\phi(\boldsymbol{\tau}^2) = [H^2A]^{[0,8-\mathbf{6}]} \cdot [H^3B \wedge [H^2C]^{[1,5+\mathbf{37}]}]^{[0,7+\mathbf{35}]} \cdot [H^1D]^{[0,3-\mathbf{2}]}$.

In Figure 7·2(a), d_{sat} decreases if there is at least one active vehicle. While two vehicles are active, whenever one of them becomes inactive, d_{sat} increases abruptly. Also, if there are no active vehicles, d_{sat} becomes undefined, shown as gaps in Figure 7·2(a). If both vehicles return to the base and $d_{sat} \neq 0$, the satisfaction loop is not re-initiated. Instead, whenever a vehicle becomes active, it continues to make progress for the uncompleted loop. Hence, the results demonstrate that a relaxed ϕ is eventually satisfied in a periodic fashion.

We also compare the proposed policy with a benchmark policy (Π_B) where a relaxation is not allowed. In other words, if ϕ can not be satisfied by the active vehicles, all vehicles return to the base. While Π_B results in only the satisfaction of ϕ, it causes a significant amount of time gaps between the satisfactions as illustrated in Figure 7·2(b). Note that 10 satisfactions of ϕ require 550 time steps whereas 10 satisfactions of the relaxed formulae are achieved in 380 time steps. The results indicate that allowing temporal relaxation of a formula increases the number of satisfactions.

7.4.2 Experimental Results

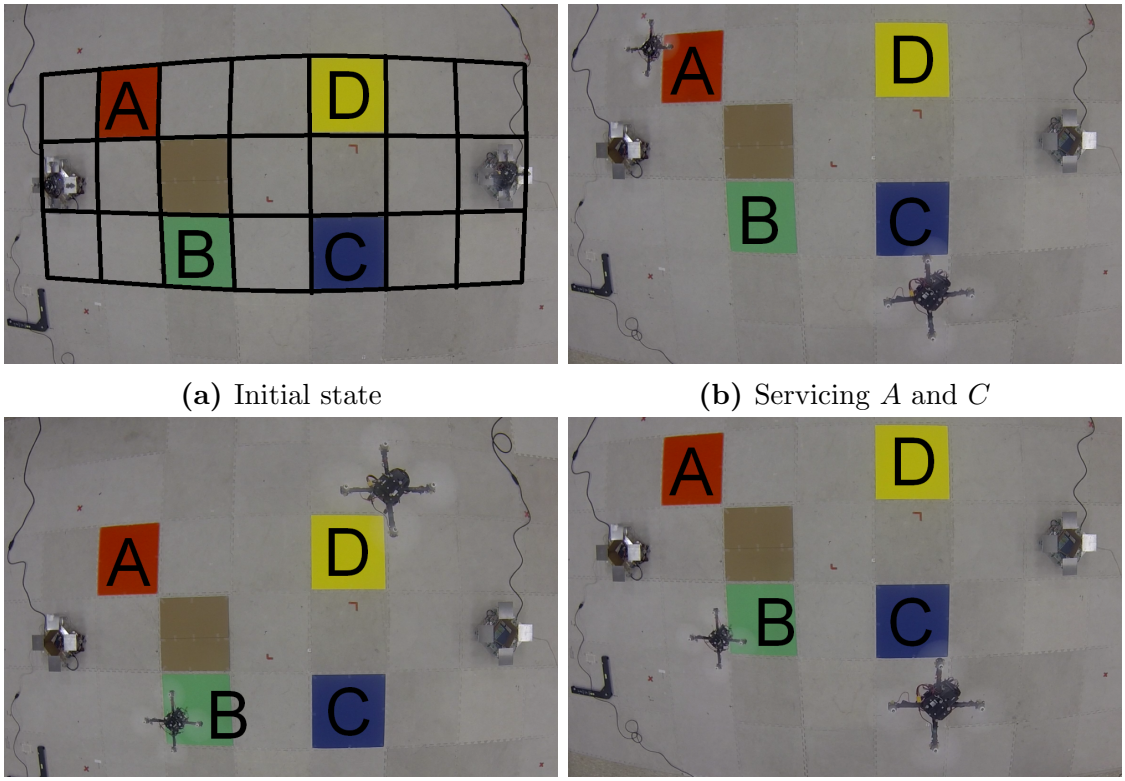
We present some preliminary results on a multi-quadrotor testbed at the BU Robotics Laboratory. The flight space is equipped with an indoor OptiTrack localization system, which tracks reflective markers mounted on K500 quadrotors from KMel Robotics. Each quadrotor is equipped with an 11.57 V 3-cell LiPo battery and custom charging gear, which allows them to automatically recharge their batteries at a

charging station. The quadrotors hover and move via local controllers, which were designed based on the differential flatness property of the quadrotors' dynamics (Leahy et al., 2014).

We consider two quadrotors and a grid environment with 4 sites and 2 charging stations as in Figure 7·3a. A quadrotor can move to any adjacent cell other than the brown cell (representing an obstacle). A unique flight altitude and charging station is assigned to each quadrotor to avoid collisions. The objective is to satisfy repeatedly

$$\phi = [H^2 A \wedge H^2 C]^{[0,8]} \cdot [H^3 B \wedge H^3 D]^{[0,7]} \cdot [[H^2 A]^{[2,6]} \vee [H^2 C]^{[1,5]}]. \tag{7.4}$$

The remaining fuel, the distance to satisfaction, and the number of active vehicles are shown in Figure 7·2(c) . Fuel in this case is interpreted as battery voltage level. In Figure 7·2(c), there exists some fluctuations in the remaining fuel due to the potential measurement errors, but a decreasing trend is observed in both the remaining fuel and the distance to satisfaction.



(a) Initial state **(b)** Servicing A and C

(c) Servicing B and D **(d)** Servicing C and end of loop

Figure 7·3: Two quadrotors in an environment with 4 sites and 2 charging stations. a the quadrotors are fully charged and docked; b the quadrotors are servicing sites A and C; c the quadrotors are servicing sites B and D; d a quadrotor is servicing site C, thus one satisfaction of (7.4) is achieved.

Chapter 8

Conclusions and Future Work

In this dissertation we considered control problems involving system that pose scalability issues. The scalability problems are due to (a) high-dimensional configuration spaces, (b) stochastic nature, and (c) multi-system structure (i.e., multi-vehicle systems). Another major problem considered is the control of systems from timed specifications. We propose a logic called *Time Window Temporal Logic* that can express rich timed temporal properties, and an automata-based framework for solving synthesis, verification, and learning problems. We then employ this framework to solve persistent multi-vehicle routing problems with charging and temporal logic constraints.

In the first part of the dissertation, we introduced a sampling-based motion planning algorithm that combines long-term temporal logic goals with short-term reactive requirements. The specification has two parts: (1) a global specification given as an LTL formula over a set of static service requests that occur at the regions of a known environment, and (2) a local specification that requires servicing a set of dynamic requests that can be sensed locally during the execution. The proposed computational framework consists of two main ingredients: (a) an off-line sampling-based algorithm for the construction of a global transition system that contains a path satisfying the LTL formula, and (b) an on-line sampling-based algorithm to generate paths that service the local requests, while making sure that the satisfaction of the global specification is not affected. Plans for future work include the implementation of these

algorithms for robots with realistic dynamics moving in complex environments, experimental trials for aerial and ground vehicles.

In the second part of the dissertation, we presented a sampling-based algorithm that generates feedback policies for stochastic systems with temporal and uncertainty constraints. The desired behavior of the system is specified using *Gaussian Distribution Temporal Logic* such that the generated policy satisfies the task specification with maximum probability. The proposed algorithm generates a transition system in the belief space of the system. A key step towards the scalability of the automata-based methods employed in the solution was breaking the *curse of history* for POMDPs. Local feedback controllers that drive the system within belief sets were employed to achieve history independence for paths in the transition system. Also contributing to the scalability of our solution is a construction procedure for an annotated product Markov Decision Process called GDTL-FIRM, where each transition is associated with a "failure probability". GDTL-FIRM captures both satisfaction and the stochastic behavior of the system. Switching feedback policies were computed over the product MDP. Lastly, we showed the performance of the computed policies in experimental trials with a ground robot tracked via camera network. The case study shows that properties specifying the temporal and stochastic behavior of systems can be expressed using GDTL and our algorithm is able to compute control policies that satisfy the specification with a given probability.

In the third part of the dissertation, we introduced a specification language called time window temporal logic (TWTL), which is a linear-time logic encoding sets of discrete-time bounded-time trajectories. We showed that TWTL has several benefits over other bounded temporal logics in terms of complexity and easiness to express and comprehend specifications. Different from other temporal logics, TWTL has an explicit concatenation operator, which enables the compact representation of serial

tasks. Such a compact representation significantly reduces the complexity of constructing the automaton for the accepting language. In this paper, we also presented temporal relaxations of TWTL formulae and provided provably-correct algorithms to construct an annotated automaton that can encode all temporal relaxations of a given TWTL formula. Moreover, we demonstrated the potential of TWTL and its relaxation on three problems related to verification, synthesis, and learning. In the verification problem, we checked whether a system can satisfy the structure of a given formula without considering its time bounds. In the synthesis problem, we found a control policy for a system that satisfies the original TWTL formula or its minimal relaxation in case of an infeaisbility. In the learning problem, we considered a data set and a template TWTL formula with parametric time bounds, and we synthesized the time parameters by minimizing the misclassification rate. Finally, we developed a Python package for the solutions of the aforementioned problems.

As future work, we plan to improve the proposed methods to make them more applicable to various areas. For example, TWTL is a good fit for statistical model checking, where the problem of learning deadlines can be modified to yield statistically robust deadline values in a template formula. The current version of the learning algorithm can find the time bounds of a given template formula (with fixed structure) from a data set. We are also working on more advanced algorithms that can infer not only the time bounds but also the structure of the template. Furthermore, we plan to improve the Python package PyTWTL by integrating automata minimization in the construction procedure in a way that (i) preserves annotation, and (ii) takes into account structure of the generated automata. Since the languages associated with TWTL formulae are finite, specialized minimization techniques may be used. For instance, one approach is to use Deterministic Finite Cover Automata (Körner, 2003; Körner, 2003; Câmpeanu et al., 2006) that can decrease the return automata's sizes

significantly with almost no additional computational cost. Other small optimizations we plan to include in PyTWTL are: (i) the case when the satisfaction of atomic propositions is assumed mutually exclusive; and (ii) preprocessing of TWTL formulae for performance improvement using AST rewriting rules. We also plan to develop AST rewriting rules to automatically transform a TWTL formula to DFW form.

Also in the third part of the dissertation, we considered a persistent vehicle routing problem involving a team of vehicles that are required to achieve a task repetitively while refueling when necessary. We expressed the task as a TWTL formula over a set of locations. We investigated two settings of the problem: (1) deterministic models for motion and charging with fixed specification; (2) stochastic charging and relaxed specifications. In the first case, we proposed a centralized automata-based framework that we show is complete and optimal with respect to two cost functions, e.g., average and maximum long-term loop time. For the second case, we proposed a hybrid control policy that decouples the refueling decision of each vehicle from the joint planning in the mission area. The proposed policy has two main benefits. First, the trajectories are computed on-line, and they are updated whenever a change occurs in the mission area. Second, if the TWTL formula is unsatisfiable, the trajectories for the active vehicles are computed by minimally relaxing the formula. To achieve this, we exploited the notion of "temporal relaxation". We demonstrated the performance of the proposed policies in both settings via simulations and experiments.

References

Agha-mohammadi, A., Chakravorty, S., and Amato, N. (2014). FIRM: Sampling-based feedback motion-planning under motion uncertainty and imperfect measurements. *The International Journal of Robotics Research*, 33(2):268–304. doi:10.1177/0278364913501564.

Aksaray, D., Leahy, K., and Belta, C. (2015). Distributed Multi-Agent Persistent Surveillance Under Temporal Logic Constraints. *Proceedings of the 5th IFAC Workshop on Distributed Estimation and Control in Networked Systems (NecSys)*, 48(22):174–179. doi:10.1016/j.ifacol.2015.10.326.

Aksaray, D., Vasile, C.-I., and Belta, C. (2016). Dynamic routing of energy-aware vehicles with temporal logic constraints. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

Alur, R., Etessami, K., La Torre, S., and Peled, D. (2001). Parametric Temporal Logic for "Model Measuring". *ACM Transactions on Computational Logic*, 2(3):388–407. doi:10.1145/377978.377990.

Asarin, E., Donzé, A., Maler, O., and Nickovic, D. (2012). Parametric Identification of Temporal Properties. In Khurshid, S. and Sen, K., editors, *Proceedings of the International Conference on Runtime Verification*, pages 147–160, Berlin, Heidelberg. Springer. doi:10.1007/978-3-642-29860-8_12.

Ayala, A. M., Andersson, S. B., and Belta, C. (2014). Formal Synthesis of Control Policies for Continuous Time Markov Processes From Time-Bounded Temporal Logic Specifications. *IEEE Transactions on Automatic Control*, 59(9):2568–2573. doi:10.1109/TAC.2014.2309033.

Bachrach, A., Prentice, S., He, R., Henry, P., Huang, A., Krainin, M., Maturana, D., Fox, D., and Roy, N. (2012). Estimation, planning, and mapping for autonomous flight using an RGB-D camera in GPS-denied environments. *The International Journal of Robotics Research*, 31(11):1320–1343. doi:10.1177/0278364912455256.

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT Press.

Bauer, A., Leucker, M., and Schallhart, C. (2011). Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14:1–14:64. doi:10.1145/2000799.2000800.

Beck, J., Prosser, P., and Selensky, E. (2003). Vehicle Routing and Job Shop Scheduling: What's the difference? In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS)*, Trento, Italy.

Belta, C. and Habets, L. (2006). Control of a class of nonlinear systems on rectangles. *IEEE Transactions on Automatic Control*, 51(11):1749 – 1759. doi:10.1109/TAC.2006.884957.

Belta, C., Isler, V., and Pappas, G. J. (2005). Discrete abstractions for robot planning and control in polygonal environments. *IEEE Transactions on Robotics*, 21(5):864–874. doi:10.1109/TRO.2005.851359.

Bender, M., Fineman, J., Gilbert, S., and Tarjan, R. (2015). A New Approach to Incremental Cycle Detection and Related Problems. *ACM Transactions on Algorithms*, 12(2):1–22. doi:10.1145/2756553.

Bertsekas, D. (2012). *Dynamic Programming and Optimal Control*. Athena Scientific, 4th edition.

Bhatia, A., Kavraki, L., and Vardi, M. (2010). Sampling-based motion planning with temporal goals. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2689–2696. doi:10.1109/ROBOT.2010.5509503.

Bombara, G., Vasile, C. I., Penedo Alvarez, F., Yasuoka, H., and Belta, C. (2016). A Decision Tree Approach to Data Classification using Signal Temporal Logic. In *Proceedings of the Hybrid Systems: Computation and Control (HSCC)*, Vienna, Austria.

Bry, A. and Roy, N. (2011). Rapidly-exploring Random Belief Trees for motion planning under uncertainty. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 723–730. doi:10.1109/ICRA.2011.5980508.

Bullo, F., Frazzoli, E., Pavone, M., Savla, K., and Smith, S. L. (2011). Dynamic Vehicle Routing for Robotic Systems. *Proceedings of the IEEE*, 99(9):1482–1504. doi:10.1109/JPROC.2011.2158181.

Burns, B. and Brock, O. (2007). Sampling-based motion planning with sensing uncertainty. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3313–3318. doi:10.1109/ROBOT.2007.363984.

Câmpeanu, C., Culik, K., I., Salomaa, K., and Yu, S. (2001). State Complexity of Basic Operations on Finite Languages. In Boldt, O. and Jrgensen, H., editors, *Automata Implementation*, volume 2214 of *Lecture Notes in Computer Science*, pages 60–70. Springer Berlin Heidelberg. doi:10.1007/3-540-45526-4_6.

Câmpeanu, C., Păun, A., and Smith, J. R. (2006). Incremental construction of minimal deterministic finite cover automata. *Theoretical Computer Science*, 363(2):135–148. doi:10.1016/j.tcs.2006.07.020.

Chen, H.-K., Hsueh, C.-F., and Chang, M.-S. (2006). The real-time time-dependent vehicle routing problem. *Transportation Research Part E: Logistics and Transportation Review*, 42(5):383–408. doi:10.1016/j.tre.2005.01.003.

Chen, Y., Tumova, J., and Belta, C. (2012). LTL Robot Motion Control based on Automata Learning of Environmental Dynamics. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Saint Paul, MN, USA. doi:10.1109/ICRA.2012.6225075.

Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L., and Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Boston, MA.

Conway, J. H. and Sloane, N. J. (1999). *Sphere Packings, Lattices and Groups*. Springer-Verlag, New York, US, 3rd edition.

Cranen, S., Groote, J. F., and Reniers, M. (2011). A linear translation from CTL* to the first-order modal μ-calculus. *Theoretical Computer Science*, 412(28):3129–3139. doi:10.1016/j.tcs.2011.02.034.

Daciuk, J. (2003). Comparison of Construction Algorithms for Minimal, Acyclic, Deterministic, Finite-State Automata from Sets of Strings. In Champarnaud, J.-M. and Maurel, D., editors, *Proceedings of the 7th International Conference Implementation and Application of Automata*, pages 255–261. Springer. doi:10.1007/3-540-44977-9_26.

Dantzig, G. B. and Ramser, J. H. (1959). The Truck Dispatching Problem. *Management Science*, 6(1):80–91. doi:10.1287/mnsc.6.1.80.

Ding, X. C., Kloetzer, M., Chen, Y., and Belta, C. (2011). Automatic Deployment of Robotic Teams. *IEEE Robotics and Automation Magazine*, 18:75–86. doi:10.1109/MRA.2011.942117.

Ding, X. C., Lazar, M., and Belta, C. (2014). LTL Receding Horizon Control for Finite Deterministic Systems. *Automatica*, 50(2):399–408. doi:10.1016/j.automatica.2013.11.030.

Dobson, A. and Bekris, K. E. (2013). Improving Sparse Roadmap Spanners. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4106–4111. doi:10.1109/ICRA.2013.6631156.

Duret-Lutz, A. (2013). Manipulating LTL formulas using Spot 1.0. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *Lecture Notes in Computer Science*, pages 442–445, Hanoi, Vietnam. Springer. doi:10.1007/978-3-319-02444-8_31.

Fainekos, G. E., Girard, A., Kress-Gazit, H., and Pappas, G. J. (2009). Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343–352. doi:10.1016/j.automatica.2008.08.008.

Gao, Y., Salomaa, K., and Yu, S. (2011). Transition Complexity of Incomplete DFAs. *Fundamenta Informaticae*, 110(1-4):143–158.

Garey, M. and Johnson, D. (1979). *Computers and Intractibility: a Guide to Theory of NP-completeness*. W.H. Freeman Co, New York.

Gastin, P. and Oddoux, D. (2001). Fast LTL to Büchi Automata Translation. In Berry, G., Comon, H., and Finkel, A., editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France. Springer. doi:10.1007/3-540-44585-4_6.

Gol, E. A. and Belta, C. (2013). Time-Constrained Temporal Logic Control of Multi-Affine Systems. *Nonlinear Analysis: Hybrid Systems*, 10:21–23. doi:10.1016/j.nahs.2013.03.002.

Guo, M. and Dimarogonas, D. V. (2015). Multi-agent plan reconfiguration under local LTL specifications. *The International Journal of Robotics Research*, 34(2):218–235. doi:10.1177/0278364914546174.

Haeupler, B., Kavitha, T., Mathew, R., Sen, S., and Tarjan, R. E. (2012). Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. *ACM Transactions on Algorithms*, 8(1):3:1–3:33. doi:10.1145/2071379.2071382.

Hagberg, A. A., Schult, D. A., and Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA.

Han, Y.-S. and Salomaa, K. (2007). State Complexity of Union and Intersection of Finite Languages. In Harju, T., Karhumki, J., and Lepist, A., editors, *Developments in Language Theory*, volume 4588 of *Lecture Notes in Computer Science*, pages 217–228. Springer Berlin Heidelberg. doi:10.1007/978-3-540-73208-2_22.

Hauser, K. (2011). *Algorithmic Foundations of Robotics IX: Selected Contributions of the Ninth International Workshop on the Algorithmic Foundations of Robotics*, chapter Randomized Belief-Space Replanning in Partially-Observable Continuous

Spaces, pages 193–209. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-17452-0_12.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Jha, S. K., Clarke, E. M., Langmead, C. J., Legay, A., Platzer, A., and Zuliani, P. (2009). A Bayesian Approach to Model Checking Biological Systems. In *Proceedings of the 7th International Conference on Computational Methods in Systems Biology*, CMSB '09, pages 218–234, Berlin, Heidelberg. Springer-Verlag. doi:10.1007/978-3-642-03845-7_15.

Jin, X., Donze, A., Deshmukh, J., and Seshia, S. (2015). Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717. doi:10.1109/TCAD.2015.2421907.

Jones, A., Schwager, M., and Belta, C. (2013). Distribution temporal logic: Combining correctness with quality of estimation. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pages 4719–4724. doi:10.1109/CDC.2013.6760628.

Kaelbling, L., Littman, M., and Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99 – 134. doi:10.1016/S0004-3702(98)00023-X.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45. doi:10.1115/1.3662552.

Karaman, S. and Frazzoli, E. (2008). Vehicle Routing Problem with Metric Temporal Logic Specifications. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pages 3953 – 3958. doi:10.1109/CDC.2008.4739366.

Karaman, S. and Frazzoli, E. (2009). Sampling-based Motion Planning with Deterministic μ-Calculus Specifications. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, Shanghai, China. doi:10.1109/CDC.2009.5400278.

Karaman, S. and Frazzoli, E. (2011a). Linear temporal logic vehicle routing with applications to multi-UAV mission planning. *International Journal of Robust and Nonlinear Control*, 21(12):1372–1395. doi:10.1002/rnc.1715.

Karaman, S. and Frazzoli, E. (2011b). Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research*, 30(7):846–894. doi:10.1177/0278364911406761.

Karaman, S. and Frazzoli, E. (2012). Sampling-based Optimal Motion Planning with Deterministic μ-Calculus Specifications. In *Proceedings of the American Control Conference (ACC)*. doi:10.1109/ACC.2012.6315419.

Kavraki, L., Svestka, P., Latombe, J., and Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580. doi:10.1109/70.508439.

Kim, K., Fainekos, G., and Sankaranarayanan, S. (2015). On the minimal revision problem of specification automata. *The International Journal of Robotics Research*. doi:10.1177/0278364915587034.

Klein, J. and Baier, C. (2006). Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363(2):182 – 195. doi:10.1016/j.tcs.2006.07.022.

Kloetzer, M. and Belta, C. (2008). A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Transactions on Automatic Control*, 53(1):287–297. doi:10.1109/TAC.2007.914952.

Kong, Z., Jones, A., Medina Ayala, A., Aydin Gol, E., and Belta, C. (2014). Temporal Logic Inference for Classification and Prediction from Data. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 273–282, New York, NY, USA. ACM. doi:10.1145/2562059.2562146.

Körner, H. (2003). On Minimizing Cover Automata for Finite Languages in O(N Log N) Time. In *Proceedings of the 7th International Conference on Implementation and Application of Automata*, CIAA'02, pages 117–127, Berlin, Heidelberg. Springer-Verlag.

Körner, H. (2003). A time and space efficient algorithm for minimizing cover automata for finite languages. *International Journal of Foundations of Computer Science*, 14(06):1071–1086. doi:10.1142/S0129054103002187.

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299. doi:10.1007/BF01995674.

Kress-Gazit, H., Fainekos, G. E., and Pappas, G. J. (2007). Where's Waldo? Sensor-based temporal logic motion planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3116–3121. doi:10.1109/ROBOT.2007.363946.

Kress-Gazit, H., Fainekos, G. E., and Pappas, G. J. (2009). Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381. doi:10.1109/TRO.2009.2030225.

Kupferman, O. and Y. Vardi, M. (2001). Model Checking of Safety Properties. *Form. Methods Syst. Des.*, 19(3):291–314. doi:10.1023/A:1011254632723.

Laporte, G. (1992). The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345–358. doi:10.1016/0377-2217(92)90192-C.

Latvala, T. (2003). Effcient model checking of safety properties. In *Proceedings of the 10th International Conference on Model Checking Software*, SPIN, pages 74–88, Berlin, Heidelberg. Springer-Verlag.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press, Cambridge, U.K. Available at http://planning.cs.uiuc.edu/.

LaValle, S. M. and Kuffner, J. J. (1999). Randomized kinodynamic planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 473–479. doi:10.1109/ROBOT.1999.770022.

Leahy, K., Jones, A., Schwager, M., and Belta, C. (2015). Distributed information gathering policies under temporal logic constraints. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pages 6803–6808. doi:10.1109/CDC.2015.7403291.

Leahy, K., Zhou, D., Vasile, C.-I., Oikonomopoulos, K., Schwager, M., and Belta, C. (2014). Provably Correct Persistent Surveillance for Unmanned Aerial Vehicles Subject to Charging Constraints. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*.

Lesser, K. and Oishi, M. (2015). Finite State Approximation for Verification of Partially Observable Stochastic Hybrid Systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 159–168, New York, NY, USA. ACM. doi:10.1145/2728606.2728632.

Lindemann, S. R. and LaValle, S. M. (2009). Simple and Efficient Algorithms for Computing Smooth, Collision-Free Feedback Laws Over Given Cell Decompositions. *The International Journal of Robotics Research*, 28(5):600–621. doi:10.1177/0278364908099462.

Livingston, S. C. and Murray, R. M. (2013). Just-in-time synthesis for motion planning with temporal logic. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*. doi:10.1109/ICRA.2013.6631298.

Livingston, S. C., Prabhakar, P., Jose, A. B., and Murray, R. M. (2013). Patching task-level robot controllers based on a local μ-calculus formula. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*. doi:10.1109/ICRA.2013.6631229.

Ma, Y., Soatto, S., Kosecka, J., and Sastry, S. S. (2003). *An Invitation to 3-D Vision: From Images to Geometric Models*. Springer-Verlag.

Maia, E., Moreira, N., and Reis, R. (2013). Incomplete Transition Complexity of Some Basic Operations. In van Emde Boas, P., Groen, F. C., Italiano, G. F., Nawrocki, J., and Sack, H., editors, *SOFSEM: Theory and Practice of Computer Science*, volume 7741 of *Lecture Notes in Computer Science*, pages 319–331. Springer Berlin Heidelberg. doi:10.1007/978-3-642-35843-2_28.

Maler, O. and Nickovic, D. (2004). Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer. doi:10.1007/978-3-540-30206-3_12.

Maly, M., Lahijanian, M., Kavraki, L. E., Kress-Gazit, H., and Vardi, M. Y. (2013). Iterative Temporal Motion Planning for Hybrid Systems in Partially Unknown Environments. In *Proceedings of the ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 353–362, Philadelphia, PA, USA. doi:10.1145/2461328.2461380.

Manna, Z. and Pnueli, A. (1981). Verification of Concurrent Programs. Part I. The Temporal Framework. Technical report, DTIC Document.

Mu, Q., Fu, Z., Lysgaard, J., and Eglese, R. (2011). Disruption management of the vehicle routing problem with vehicle breakdown. *Journal of the Operational Research Society*, 62(4):742–749. doi:10.1057/jors.2010.19.

Papadimitriou, C. and Tsitsiklis, J. (1987). The complexity of Markov Decision Processes. *Mathematics of Operations Research*, 12(3):441–450. doi:10.1287/moor.12.3.441.

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages* . Pragmatic Bookshelf.

Patil, S., Kahn, G., Laskey, M., Schulman, J., Goldberg, K., and Abbeel, P. (2015). *Algorithmic Foundations of Robotics XI: Selected Contributions of the Eleventh International Workshop on the Algorithmic Foundations of Robotics*, chapter Scaling up Gaussian Belief Space Planning Through Covariance-Free Trajectory Optimization and Automatic Differentiation, pages 515–533. Springer, Cham. doi:10.1007/978-3-319-16595-0_30.

Pavone, M., Bisnik, N., Frazzoli, E., and Isler, V. (2009). A stochastic and dynamic vehicle routing problem with time windows and customer impatience. *Mobile Networks and Applications*, 14(3):350–364. doi:10.1007/s11036-008-0101-1.

Pineau, J., Gordon, G., and Thrun, S. (2003). Point-based Value Iteration: An Anytime Algorithm for POMDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico.

Prentice, S. and Roy, N. (2009). The belief roadmap: Efficient planning in belief space by factoring the covariance. *The International Journal of Robotics Research*. doi:10.1177/0278364909341659.

Puterman, M. L. (2014). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, NY, USA.

Raman, V., Lignos, C., Finucane, C., Lee, K. C. T., Marcus, M., and Kress-Gazit, H. (2013). Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language. In *Proceedings of the Robotics: Science and Systems Conference (RSS)*, Berlin, Germany.

Reyes Castro, L., Chaudhari, P., Tumova, J., Karaman, S., Frazzoli, E., and Rus, D. (2013). Incremental sampling-based algorithm for minimum-violation motion planning. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pages 3217–3224. doi:10.1109/CDC.2013.6760374.

Smith, S., Tumova, J., Belta, C., and Rus, D. (2011). Optimal Path Planning for Surveillance with Temporal Logic Constraints. *The International Journal of Robotics Research*, 30(14):1695–1708. doi:10.1177/0278364911417911.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265. doi:10.1287/opre.35.2.254.

Sundar, K. and Rathinam, S. (2014). Algorithms for routing an unmanned aerial vehicle in the presence of refueling depots. *IEEE Transactions on Automation Science and Engineering*, 11(1):287–294. doi:10.1109/TASE.2013.2279544.

Svorenova, M., Cerna, I., and Belta, C. (2013). Optimal control of MDPs with temporal logic constraints. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pages 3938–3943. doi:10.1109/CDC.2013.6760491.

Talata, I. (1998). Exponential Lower Bound for the Translative Kissing Number of d-Dimensional Convex Bodies. *Discrete & Computational Geometry*, 19:447–455. doi:10.1007/PL00009362.

Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press.

Tkachev, I. and Abate, A. (2013). Formula-free Finite Abstractions for Linear Temporal Verification of Stochastic Hybrid Systems. In *Proceedings of the 16th Int. Conference on Hybrid Systems: Computation and Control (HSCC)*, Philadelphia, PA. doi:10.1145/2461328.2461372.

Toth, P. and Vigo, D., editors (2001). *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Tumova, J., Hall, G. C., Karaman, S., Frazzoli, E., and Rus, D. (2013a). Least-violating control strategy synthesis with safety rules. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 1–10, New York, NY, USA. ACM. doi:10.1145/2461328.2461330.

Tumova, J., Marzinotto, A., Dimarogonas, D., and Kragic, D. (2014). Maximally satisfying ltl action planning. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1503–1510. doi:10.1109/IROS.2014.6942755.

Tumova, J., Reyes-Castro, L., Karaman, S., Frazzoli, E., and Rus, D. (2013b). Minimum-violating planning with conflicting specifications. In *Proceedings of the American Control Conference (ACC)*. doi:10.1109/ACC.2013.6579837.

Ulusoy, A., Marrazzo, M., and Belta, C. (2013a). Receding Horizon Control in Dynamic Environments from Temporal Logic Specifications. In *Proceedings of the Robotics: Science and Systems Conference (RSS)*.

Ulusoy, A., Marrazzo, M., Oikonomopoulos, K., Hunter, R., and Belta, C. (2013b). Temporal Logic Control for an Autonomous Quadrotor in a Nondeterministic Environment. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. doi:10.1109/ICRA.2013.6630596.

Ulusoy, A., Smith, S. L., Ding, X. C., Belta, C., and Rus, D. (2013c). Optimality and Robustness in Multi-Robot Path Planning with Temporal Logic Constraints. *International Journal of Robotics Research*, 32(8):889–911. doi:10.1177/0278364913487931.

van den Berg, J., Abbeel, P., and Goldberg, K. (2011). LQG-MP: Optimized path planning for robots with motion uncertainty and imperfect state information. *The International Journal of Robotics Research*, 30(7):895–913. doi:10.1177/0278364911406562.

Vasile, C. and Belta, C. (2013). Sampling-Based Temporal Logic Path Planning. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tokyo. doi:10.1109/IROS.2013.6697051.

Vasile, C. and Belta, C. (2014a). Reactive Sampling-Based Temporal Logic Path Planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong.

Vasile, C.-I., Aksaray, D., and Belta, C. (2016). Time Window Temporal Logic. *arXiv preprint arXiv:1602.04294v1*.

Vasile, C.-I. and Belta, C. (2014b). An Automata-Theoretic Approach to the Vehicle Routing Problem. In *Proceedings of the Robotics: Science and Systems Conference (RSS)*, Berkeley, California, USA.

Vilenkin, N. Y. (1971). *Combinatorics*. Academic Press, NY, USA.

Vitus, M. P. and Tomlin, C. J. (2011). Closed-loop belief space planning for linear, gaussian systems. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2152–2159. doi:10.1109/ICRA.2011.5980257.

Wongpiromsarn, T., Topcu, U., and Murray, R. M. (2009). Receding Horizon Temporal Logic Planning for Dynamical Systems. In *Proceedings of the IEEE Conference on Decision and Control (CDC)*, pages 5997–6004. doi:10.1109/CDC.2009.5399536.

Wongpiromsarn, T., Topcu, U., and Murray, R. M. (2010). Receding horizon control for temporal logic specifications. In *Proceedings of the 13th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 101–110. ACM. doi:10.1145/1755952.1755968.

Wood, D. R. (2004). Bounded degree acyclic decompositions of digraphs. *Journal of Combinatorial Theory, Series B*, 90(2):309–313. doi:10.1016/j.jctb.2003.08.004.

Yang, H., Hoxha, B., and Fainekos, G. (2012). Querying Parametric Temporal Logic Properties on Embedded Systems. In Nielsen, B. and Weise, C., editors, *Proceedings of the IFIP WG 6.1 International Conference on Testing Software and Systems*, pages 136–151, Berlin, Heidelberg. Springer. doi:10.1007/978-3-642-34691-0_11.

Zamani, M., Esfahani, P. M., Majumdar, R., Abate, A., and Lygeros, J. (2014). Symbolic Control of Stochastic Systems via Approximately Bisimilar Finite Abstractions. *IEEE Transactions on Automatic Control*, 59(12):3135–3150. doi:10.1109/TAC.2014.2351652.

Zhou, D. and Schwager, M. (2014). Vector Field Following for Quadrotors using Differential Flatness. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. doi:10.1109/ICRA.2014.6907828.

CURRICULUM VITAE

Cristian-Ioan Vasile

Contact Information

001-617-763-1251
cvasile@bu.edu
cristian.ioan.vasile@gmail.com
www.cristianvasile.com

Division of Systems Engineering
15 Saint Mary's Street
Brookline, MA 02446

Education

2012–current	**PhD Candidate**, Hybrid and Networked Systems (Hyness) Group, BU Robotics Lab, Division of Systems Engineering, College of Engineering, Boston University, **Advisor: Prof PhD Calin Belta** *Systems Engineering*
2011–2014	**PhD**, Department of Automatic Control and Systems Engineering, Politehnica University of Bucharest, **Advisor: Prof PhD Ioan Dumitrache** *Control Engineering*
2009–2011	**Master**, Department of Automatic Control and Systems Engineering, Politehnica University of Bucharest, *10.00* *Intelligent Control Systems*
2009–2010	**Certificate**, Department of Teacher Training, Politehnica University of Bucharest, *10.00* *Pedagogical Studies Graduate Program – Level 2 (Advanced)*
2005–2009	**Bachelor**, Faculty of Automatic Control and Computers, Politehnica University of Bucharest, *9.49* *Conputer Science, focus on Embedded Systems*
2005–2009	**Certificate**, Department of Teacher Training, Politehnica University of Bucharest, *10.00* *Pedagogical Studies – Level 1 (annex to bachelor diploma)*

Research Fellowships and Summer Schools

11–18 March 2012	*Research Fellowship*, Faculty of Philosophy and Science in Opava, Silesian University in Opava, Czech Republic – reference: Prof PhD Jozef Kelemen
5–7 September 2011	*First International School on Biomolecular and Biocellular Computing*, Osuna, Spain – awarded tuition, travel and accommodation grant – reference: Prof PhD Miguel A. Gutiérrez, ISBBC2011
24 September – 1 October 2010	*Neural Dynamics Approaches to Cognitive Robotics*, Ruhr-Universität, Bochum, Germany – awarded tuition, travel and accommodation grant – reference: Prof PhD Gregor Schöner, Neural Dynamics 2010
22–26 July 2010	1st *Cooperative Cognitive Control for Autonomous Underwater Vehicles*, Jacobs University, Bremen, Germany – awarded tuition and accommodation grant – reference: Prof PhD Kaustubh Pathak and Prof PhD Andreas Birk, Co3-AUVs 2010

Awards

NSF Student Travel Award	IEEE International Conference on Robotics and Automation (ICRA) 2014 in Hong Kong, China.
SE PhD Student Travel Award	Systems Engineering Division, Boston University: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 2013 in Tokyo, Japan; European Control Conference (ECC) 2015 in Linz, Austria.
BU Dean's Fellow	2012–2013, from the Division of Systems Engineering, College of Engineering, Boston University.
Roberto Rocca Scholarship	Merit-based award for academic excellence and leadership, national selection process, Roberto Rocca Educational Program, TenarisSilcotub.
Academic Scholarship	Merit-based scholarship during my undergraduate and graduate studies (5.5 years), performance evaluated each semester.
Award for Scientific Publication	Awards for publication by the Romanian National Council of Scientific Research for the papers in *Applied Soft Computing* journal and *BMC Bioinformatics* journal.
1st prize at Student Scientific Session	1st prize at Student Scientific Session in the Automatic Control and Systems Engineering section, Politehnica University of Bucharest, Romania, 2007 for the implementation of a vision system for a garbage collector robot and for the paper: "Ana Pavel, **Cristian Ioan Vasile**, *Artificial vision system of the ReMaster robot using the CMUCam2+ camera*"

Publications

Journal Articles

1. **Vasile Cristian Ioan**, Aksaray Derya, and Belta Calin. Time Window Temporal Logic. *Theoretical Computer Science*, page (submitted).

2. **Vasile Cristian Ioan**, Schwager Mac, and Belta Calin. Translational and Rotational Invariance in Networked Dynamical Systems. *IEEE Transactions on Control of Network Systems*, page (submitted).

3. Leahy Kevin, Zhou Dingjiang, **Vasile Cristian Ioan**, Oikonomopoulos Konstantinos, Schwager Mac, and Belta Calin. Persistent Surveillance for Unmanned Aerial Vehicles Subject to Charging and Temporal Logic Constraints. *Autonomous Robots*, page 1–16, 2016. doi:10.1007/s10514-015-9519-z.

4. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, and Dumitrache Ioan. Improving the universality results of Enzymatic Numerical P Systems. *International Journal of Computer Mathematics (special issue: Membrane Computing)*, 90(4), February 2013. if=0.589, doi: 10.1080/00207160.2012.748897.

5. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, Dumitrache Ioan, and Păun Gheorghe. On the Power of Enzymatic Numerical P Systems. *Acta Informatica*, 49(6):395–412, September 2012. if=0.809, doi:10.1007/s00236-012-0166-y.

6. Buiu Cătălin, **Vasile Cristian Ioan**, and Arsene Octavian. Development of membrane controllers for mobile robots. *Information Sciences*, 187:33–51, March 2012. if=2.833, doi:10.1016/j.ins.2011.10.007.

7. Pavel Ana Brânduşa and **Vasile Cristian Ioan**. PyElph – a Software Tool for Gel Images Analysis and Phylogenetics. *BMC Bioinformatics*, 13(9), January 2012. if=3.03, doi:10.1186/1471-2105-13-9 (Open Access).

8. **Vasile Cristian Ioan** and Buiu Cătălin. A software system for collaborative robotics applications and its application in particle swarm optimization implementations. *Applied Soft Computing*, 11(8):5498–5507, December 2011. if=2.084, doi:10.1016/j.asoc.2011.05.009.

9. **Vasile Cristian Ioan** and Constantinescu Alexandru. On the quotient criterion. *Gazeta Matematică*, CX(9):420–422, 2005. in Romanian.

Conference Articles

1. **Vasile Cristian Ioan**, Leahy Kevin, Cristofalo Eric, Jones Austin, Schwager Mac, and Calin Belta. Control in Belief Space with Temporal Logic Specifications. In *IEEE Conference on Decision and Control (CDC)*, page (submitted), December 2016.

2. **Vasile Cristian Ioan**, Vaidyanathan Prashant, Madsen Curtis, Densmore Douglas, and Calin Belta. Compositional Signal Temporal Logic with Applications to Synthetic Biology. In *IEEE Conference on Decision and Control (CDC)*, page (submitted), December 2016.

3. **Vasile Cristian Ioan**, Pavel Ana Brandusa, and Dumitrache Ioan. Variable Structure Controllers Using Hybrid Numerical P Systems. In *IEEE Conference on Decision and Control (CDC)*, page (submitted), December 2016.

4. Bombara Giuseppe, **Vasile Cristian Ioan**, Penedo Alvarez Francisco, and Belta Calin. A Decision Tree Approach to Data Classification using Signal Temporal Logic. In *Hybrid Systems: Computation and Control (HSCC)*, page (accepted), Vienna, Austria, April 2016.

5. Aksaray Derya, **Vasile Cristian Ioan**, and Belta Calin. Dynamic Routing of Energy-Aware Vehicles with Temporal Logic Constraints. In *IEEE International Conference on Robotics and Automation (ICRA)*, page (accepted), 2016.

6. **Vasile Cristian Ioan**, Schwager Mac, and Belta Calin. SE(N) Invariance in Networked Systems. In *European Control Conference (ECC)*, pages 186–191, Linz, Austria, July 2015. doi:10.1109/ECC.2015.7330544.

7. **Vasile Cristian Ioan** and Belta Calin. An Automata-Theoretic Approach to the Vehicle Routing Problem. In *Robotics: Science and Systems Conference (RSS)*, pages 1–9, Berkeley, California, USA, July 2014. link.

8. Leahy Kevin, Zhou Dingjiang, **Vasile Cristian Ioan**, Oikonomopoulos Konstantinos, Schwager Mac, and Belta Calin. Provably Correct Persistent Surveillance for Unmanned Aerial Vehicles Subject to Charging Constraints. In *International Symposium on Experimental Robotics (ISER)*, Marrakech, Essaouira, Morocco, June 2014.

9. **Vasile Cristian Ioan** and Belta Calin. Reactive Sampling-Based Temporal Logic Path Planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4310–4315, Hong Kong, China, June 2014. doi:10.1109/ICRA.2014.6907486.

10. **Vasile Cristian Ioan** and Belta Calin. Sampling-Based Temporal Logic Path Planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4817–4822, Tokyo, Japan, November 2013. doi:10.1109/IROS.2013.6697051.

11. Pavel Ana Brânduşa, **Vasile Cristian Ioan**, and Dumitrache Ioan. Robot localization implemented with enzymatic numerical P systems. In *Proc. of the Living Machines 2012: The International Conference on Biomimetic and Biohybrid Systems*, volume 7375 of *Lecture Notes in Computer Science*, pages 204–215, Barcelona, Spain, July 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31525-1_18.

12. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, and Dumitrache Ioan. Improving the universality results of Enzymatic Numerical P Systems. In *Proc. of the 10th Brainstorming Week on Membrane Computing*, pages 215–228, Seville, Spain, February 2012. link.

13. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, Dumitrache Ioan, and Păun Gheorghe. Numerical P Systems. In *Proc. of the 10th Brainstorming Week on Membrane Computing*, pages 26–29, Seville, Spain, February 2012. collective paper: Research Topics in Membrane Computing: After CMC 12, Before BWMC 10, Eds. Gheorghe M., Paun Gh., Perez-Jimenez M.J.

14. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, Dumitrache Ioan, and Kelemen Jozef. Implementing obstacle avoidance and follower behaviors on Koala robots using Numerical P Systems. In *Proc. of the 10th Brainstorming Week on Membrane Computing*, pages 207–214, Seville, Spain, February 2012. link.

15. Buiu Cătălin, Pavel Ana Brânduşa, **Vasile Cristian Ioan**, and Dumitrache Ioan. Perspectives of using membrane computing in the control of mobile robots. In *Proc. of the Beyond AI - Interdisciplinary Aspect of Artificial Inteligence Conference*, pages 21–26, Pilsen, Czech Republic, December 2011. link.

16. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, and Buiu Cătălin. Integrating human swarm interaction in a distributed robotic control system. In *Proc. of the IEEE 7th Annual IEEE Conference on Automation Science and Engineering (CASE)*, pages 743–748, Trieste, Italy, August 2011. doi:10.1109/CASE.2011.6042493.

17. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, and Buiu Cătălin. Chidori - a bio-inspired cognitive architecture for collective robotics applications. In *Proc. of the IFAC Workshop on Intelligent Control Systems*, pages 52–57, Sinaia, Romania, September 2010. link.

18. Pavel Ana Brânduşa, **Vasile Cristian Ioan**, and Buiu Cătălin. Cognitive vision system for an ecological mobile robot. In *Proc. of the 13th International Symposium on System Theory, Automation, Robotics, Computers, Informatics, Electronics and Instrumentation (SINTES)*, volume 1, pages 267–272, Craiova, Romania, October 2007. link.

Books and Chapters

1. Kevin Leahy, Dingjiang Zhou, **Cristian-Ioan Vasile**, Konstantinos Oikonomopoulos, Mac Schwager, and Calin Belta. Provably correct persistent surveillance for unmanned aerial vehicles subject to charging constraints. In M. Ani Hsieh, Oussama Khatib, and Vijay Kumar, editors, *Experimental Robotics*, volume 109 of *Springer Tracts in Advanced Robotics*, pages 605–619. Springer International Publishing, 2016. isbn: 978-3-319-23777-0, link.

2. Pavel Ana Brânduşa, **Vasile Cristian Ioan**, and Dumitrache Ioan. *Membrane computing in robotics*, volume 4 of *Topics in Intelligent Engineering and Informatics (special issue: Beyond Artificial Intelligence)*, pages 125–136. Springer, 2013. isbn-13: 978-3642344213.

3. Pavel Ana Brânduşa, **Vasile Cristian Ioan**, and Buiu Cătălin. *Biomathematics and Bioinformatics – Concepts and Applications*. Editura Universitară, Bucharest, Romania, 2011. isbn: 978-606-591-178-9, in Romanian.

4. Buiu Cătălin, Pavel Ana Brânduşa, and **Vasile Cristian Ioan**. *Cognitive Robots – Bio-inspired Applications*. Editura Universitară, Bucharest, Romania, 2010. isbn: 978-973-749-835-9, in Romanian.

5. Pavel Ana Brânduşa and **Vasile Cristian Ioan**. *Cognitive Robots – Concepts, Architecures, Applications*, chapter II: Robots with cognitive vision. Case study – ReMaster One robot, pages 35–97. Editura Universitară, Bucharest, Romania, 2008. isbn: 978-973-749-443-6, in Romanian.

Posters

1. **Vasile Cristian Ioan** and Belta Calin. Reactive Sampling-Based Temporal Logic Path Planning. In *5th Workshop on Formal Methods for Robotics and Automation*, page Poster, Berkeley, CA, USA, July 2014. link.

2. **Vasile Cristian Ioan**, Pavel Ana Brânduşa, Arsene Octavian, Popescu Nirvana, and Buiu Cătălin. Human-swarm interface design and new control techniques for swarms autonomous mobile robots. In *Proc of the 4th International Conference on Cognitive Systems (CogSys)*, page Poster, ETH Zurich, Switzerland, January 2010. link.

Talks and demonstrations

Talks

8 December 2015	Temporal Logic Planning and Inference, Distributed Robotics Laboratory, MIT.
5 September 2011	"Membrane Controllers for Mobile Robots" at the First International School on Biomolecular and Biocellular Computing, Osuna, Spain – reference: Prof PhD Miguel A. Gutiérrez, ISBBC2011
18 June 2011	"Modeling and simulation of human HIV-1 gp120 envelope glycoprotein" at the IBM High Performance Scientific Computing Workshop, Bucharest, Romania
10 November 2010	"Particle Swarm Optimization and its applications in collaborative robotics" at the Laboratory of Natural Computing and Robotics

Demonstrations

19–20 February 2011	"Chidori Architecture – Distributed Swarm Control System and User Interface" poster and stand at the Artificial Intelligence – Multi-Agent Systems (AI-MAS) Winter Olympics, Politehnica University of Bucharest, Bucharest, Romania

Membership and Community Service

Membership	IEEE Student Member, IEEE RAS Member, EuCogIII member, Founding member of Romanian Robotics Education Initiative (RREI)
Reviewer	International Journal of Robotics Research, IEEE Robotics and Automation Letters, IEEE Transactions on Automation Science and Engineering, Theoretical Computer Science Journal, Discrete Event Dynamic Systems, IEEE Transactions on NanoBioscience, Applied Soft Computing Journal, Sensors Journal, Robotics: Science and Systems Conference (RSS 2016), IEEE International Conference on Robotics and Automation (ICRA 2015, 2016), IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2016), ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2016), IEEE Conference on Decision and Control (CDC 2014, 2015), IFAC Workshop on Distributed Estimation and Control in Networked Systems(NecSys 2015), IFAC Workshop on Intelligent Control Systems (WICS 2010)
Organizer	IFAC Workshop on Intelligent Control Systems (WICS, 2010)
Judge Advisor	CEESA First Tech Challenge robotics competition, American International School of Bucharest, April 2011 and March 2012

Interests

Robotics	formal methods, path planning, swarm robotics, distributed and decentralized control
Control engineering	correct-by-construction control strategies, temporal logics, sampling based algorithms, incremental computing
Other	bioinformatics, graph coloring

Research Experience

2013–current **Research Assistant**, Hybrid and Networked Systems (HyNeSs) Group, BU Robotics Lab, Division of Systems Engineering, College of Engineering, Boston University,

2007–2012 **Volunteer Researcher**, Laboratory of Natural Computing and Robotics, Politehnica University of Bucharest,

Teaching Experience

2011–2012 **Teaching Assistant**, Politehnica University of Bucharest,
Laboratory Classes:
- *Robotics and Virtual Reality (Spring 2012);*
- *Control Engineering (Spring 2012);*
- *Programming real-time applications (Spring 2012);*
- *Diagnosis and Decision Techniques (Spring 2012);*
- *Artificial Intelligence (Fall 2011).*

2010–2011 **Associate Teaching Assistant**, Politehnica University of Bucharest,
Laboratory Classes:
- *Robotics and Virtual Reality (Spring 2010, Spring 2011);*
- *Control Engineering (Spring 2011);*
- *Cognitive Robotics (winter 2010);*
- *Intelligent Multi-agent Systems for Ambient Assistance (winter 2010).*

2009–2010 **Volunteer Teaching Assistant**, Politehnica University of Bucharest,
Laboratory Classes:
- *Robotics and Virtual Reality (Spring 2009);*
- *Microprocessor Based Design (Spring 2009, Spring 2010).*

Projects

Current

2016–	Temporal Logic Planning for Support by Fire Operations in Uncertain and Adversarial Environments
2013–	Persistent Vehicle Routing Problem with Temporal Logic and Charging Constraints
2013–	Sampling-Based Motion Planning for Stochastic Systems with Distribution Temporal Logic
2012–	Reactive Sampling-Based Path Planning with Temporal Logic Specifications
2015–	Compositional Signal Temporal Logic with Applications to Synthetic Biology
2014–	Translational and Rotational Invariance in Networked Systems
2015–	Data-driven Inference of Temporal Logic Specifications
2014–	Time Window Temporal Logic
2014–	Bio-Electrical Cell Networks
2012–	Hybrid Numerical P Systems. Controllers with Time-Varying Structure
2011–	PyElph – open source software tool for gel image analysis and phylogenetics

Finished

2009–2012	Chidori, a distributed multi-agent control architecture for multi-robot systems using JADE
2009–2011	Robot controllers modeled with Numerical P Systems (NPS) and Enzymatic NPS
2009–2010	PSO based search algorithm of a target in an unknown environment using Khepera III and e-puck robots
2009	Software package for working with Khepera and e-puck robots
2009	Fuzzy filters for noise reduction in images
2008	Design and construction of an autonomous robot (JBot)
2008	Compiler for the Cool didactic programming language
2008	Development of a ssh client for the Android platform
2008	Didactic processor on a FPGA, Sparten3E
2006–2007	Design and construction of ReMaster, an autonomous service robot
2006–2007	Artificial vision system for the ReMaster robot

Skills

Computer skills

Programming languages	Python, C, Java, R, Matlab, PHP, SQL, Bash
Frameworks	ROS, ANTLR, MPI, wxPython, matplotlib, numpy, scipy

Languages

Romanian	native language
English	**Advanced** TOEFL Score: 111 – R:30, L:30, S: 23, W: 28
German	**Advanced** German Certificate "Zertifikat Deutsch", Goethe Institute (98%)

Other skills

Driving license	category B
Artistic	violin and music theory, received 9.95/10 in the national examination "Capacitate" (2001)