Combining reactive synthesis and motion planning to control complex systems

Rostan Tabet
(supervisor: Philipp Schlehuber-Caissier)

Technical Report n°2205, January 2024
revision 03c55577

We address the problem of combining reactive synthesis and motion planning to control a system with complex dynamics according to a high-level specification. From a description of the problem, we synthesize a controller that provides a solution. We then map this controller into the real world using motion planning.

Nous nous intéressons au problème de la combinaison de la synthèse réactive et de la planification de mouvement pour contrôler un système avec une dynamique complexe, selon une spécification de haut niveau. À partir d'une description du problème, nous synthétisons un contrôleur qui fournit une solution. Nous transposons ensuite ce contrôleur dans le monde réel en utilisant la planification de mouvement.

Keywords
Reactive synthesis, LTL, GR(1), Game theory, Motion planning, RRT

Laboratoire de Recherche et Développement de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
rostan.tabet@epita.fr – http://www.lre.epita.fr/

Copying this document

Contents

Motivations

Controlling robots to perform complex tasks is often achieved by programmers in a manner that is both time-consuming and error-prone. For example, machines in manufacturing cells are reconfigurable, but the code isn't, which means that the costly process of writing it must be repeated each time. Using formal methods to automate this process would shift the programming paradigm for this problem from an imperative to a logical one. This approach would provide the ability to only focus on a high-level problem description while producing a correct-by-construction low-level solution.

The reactive synthesis problem, also known as CHURCH*'s problem* [3], consists in synthesizing a circuit from a logical specification. There are many choices for a logical formalism of the specification. Temporal logics are observed to be adequate to express most problems. The given specification then needs to be transformed into a game by choosing one of the many available strategies. One possible method is, for example, to start from an LTL specification φ and then apply the following transforms: [4]

- Translate φ to a non-deterministic generalized Büchi automaton, which is used to produce a non-deterministic Büchi automaton (nBA). On the worst case, this may produce exponentially many states in the size of φ.
- Translate this nBA into a deterministic parity automaton. On the worst case, this can also cause an exponential blowup on the size of the nBA.

The problem is that the double translation requires an exponential computation time, as a function of the input specification size.

The high complexity makes this approach not amenable to most industrial-sized problems and other methods that use LTL specifications stay in the same complexity class. To fall into a lower complexity class, a solution is to only consider a fragment of LTL. In the past years, work has been done to identify fragments that improve scalability while allowing to be expressive enough to represent real-world problems [1]. The one we are interested in is called *Generalized Reactive(1)*, or GR(1) [2] and requires a polynomial number of steps to synthesize a controller.

However, while the synthesized controller is discrete, the real world is continuous. The synthesis abstracts the robot's displacement from a high level so we need motion planning to represent this from a lower level. To obtain an algorithm that ensures termination, our approach is based on guided trial and error. We implement a closed loop between the motion planner and the synthesizer: the synthesizer asks for new paths for the motion planner and updates the specification whenever they cannot be computed sufficiently fast. Whenever the motion planner fails to compute a path, it is penalized when synthesizing a new controller. This makes sure that different controllers are generated each time.

In addition, because we are interested in infinite behaviors, it is natural to expect from the motion planner to produce closed paths on which the robot can then loop. This however comes back to the problem of point-to-point motion planning which is difficult to handle by the motion planning algorithms we use. We thus introduce a mechanism that will force connect the generated paths in order to create loops.

This work is made in collaboration with the ISIR.

Chapter 1

Reactive synthesis

Reactive synthesis consists in automatically generating a correct-by-construction state machine from high-level specifications. This chapter introduces the language we use to write specifications as well as the process that allows to go from such a specification to a controller.

1.1 Linear temporal logic

Linear Temporal Logic (LTL) is an extension of the Boolean logic with temporal operators. LTL is a popular formalism used for model checking, formal verification but also in contexts where we need to describe dynamic behaviors that can be modeled in discrete time such as robotics. The majority of the work on synthesis for robots indeed uses temporal logic [8].

1.1.1 Syntax

LTL formulas can be constructed as follows, where p denotes a propostional variable:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

$\neg$ can be read as "not" and $\vee$ as "or". $\bigcirc$ and $\mathcal{U}$ are temporal operators and can be read respectively as "next" and "until". Their semantics is described in the next section.

1.1.2 Semantics

A ω-word is an infinite sequence $w = a_0a_1a_2\dots$ of sets of propositional variables. For $i \geq 0$, the satisfaction relation $\models$ between a word w and a LTL formula is formally defined as follows:

- $w, 0 \models p$ if $p \in a_0$
- $w, i \models \neg\varphi$ if $w \not\models \varphi$

- $w, i \models \varphi \vee \psi$ if $w, i \models \varphi$ or $w, i \models \psi$
- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \psi)$
- $w, i \models \bigcirc\varphi$ if $w, i+1 \models \varphi$, *"φ must hold at the next step"*
- $w, i \models \varphi\,\mathcal{U}\,\psi$ if $\exists\, j \geq 0, w, j \models \psi$ and $\forall\, k \in [\![i, j]\!], w, k \models \varphi$, *"$\varphi$ must hold until ψ is true"*

$w, i \models \varphi$ can be read as "φ hold at position i of w". When $i = 0$, it can simply be written $w \models \varphi$.

We use the usual abbreviations $\rightarrow$ and $\leftrightarrow$, as well as the usual definitions for `true` and `false`:

- $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- $\texttt{true} \equiv p \vee \neg p$
- $\texttt{false} \equiv \neg\texttt{true}$

For the modal temporal operators, we use the abbreviations $\Diamond$ (eventually) and $\Box$ (globally) defined as follows:

- $\Diamond\varphi \equiv \texttt{true}\,\mathcal{U}\,\varphi$, *"$\varphi$ must eventually be true"*
- $\Box\varphi \equiv \neg\Diamond\neg\varphi$, *"$\varphi$ must always be true"*

We can then combine those operators to create formulas such as $\Box\Diamond\varphi$ (read "globally eventually φ") which means that φ must be true infinitely often.

Note There exists another notational convention for the temporal operators. Instead of $\bigcirc$, $\Box$ and $\Diamond$ the notations X, G and F are sometimes used. For the rest of this report, only the former is used.

1.2 Games

We model our problems as two-players games between a system that we want to control and an environment that is potentially adversarial, and therefore treated as such to capture the worst case. The synthesis problem then boils down to finding a winning strategy for the system. In the next sections, we focus on precisely defining those notions.

1.2.1 Explicit definition

Definition 1.2.1. A game graph is defined as a tuple $\mathcal{G} = (\mathcal{A}, \mathit{Win})$ [11]. It is composed of an arena $\mathcal{A}$ that can be seen as a graph, and a winning condition *Win*.

The main component of a game is its arena. It describes the rules the players have to follow and how they interact. It is a directed graph where each vertex is assigned to one of the players. When a play starts, a token is placed on a starting vertex of the arena. Then, dependending on who owns the vertex, the system or environment will move the token to one of the connected vertices. If the arena is a bipartite graph, the game is called an alternating game.

Definition 1.2.2. An arena is a tuple $\mathcal{A} = (V, V_e, V_s, E)$ where:

- $V = V_e \sqcup V_s$ is the set of all vertices [1].
- V_e and V_s are the sets of vertices controlled respectively by the environment and the system.
- $E \subseteq V \times V$ is the set of all edges.

Figure 1.2.1 depicts an arena where the environment vertices are represented by circles and system vertices by diamonds.

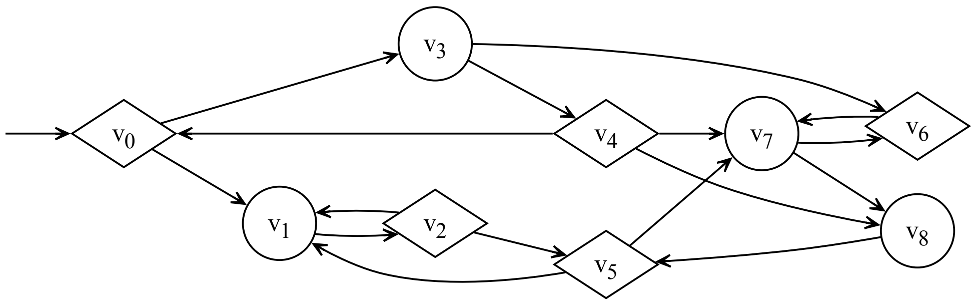


Figure 1.2.1: Arena representation generated with SPOT

If the arena has no deadlocks, after infinitely many token moves accross the arena, the two players have produced an infinite path. Such a path is called a play and is defined a follows.

Definition 1.2.3. A *play* on an $\mathcal{A}$ is a sequence $\rho = \rho_0\rho_1\rho_2\ldots \in V^\omega$ such that, for all postive integers n, $(p_n, p_{n+1}) \in E$.

[1] $\sqcup$ is used to express the disjoint union of sets

To describe how each player moves the token, the notion of strategy is introduced. The strategy defines how a player makes decisions and choses a certain edge depending on the current position on the arena, and potentially on the previous moves. A strategy that does not depend on the previous moves is called a *positional strategy*.

Definition 1.2.4. A *strategy* for the player controlling a set of vertices U is a function $\sigma : V^*p \times U \longrightarrow V$ such that, for every $w \in V^*$ and for every u in U, $\sigma(wu) = v$ implies that $(u, v) \in E$

Now that the notion of strategy is introduced, the next question is how to determine the winner of a game. A winning condition is a subset of all possible arena plays that contain the plays said to be *winning*. The system wins iff the play is in the winning condition. A winning strategy is thus one that only produces plays that are in the winning condition.

Definition 1.2.5. A winning condition is a set *Win* $\subset V^\omega$. A play ρ is winning for the system iff $\rho \in$ *Win*.

We are therefore interested in finding a winning strategy. This problem can be reduced to finding a *winning region*.

Definition 1.2.6. The *winning region* $W_s(\mathcal{G})$ (resp. $W_e(\mathcal{G})$) of the system (resp. the environment) is the set of vertices from which it has a winning strategy.

To construct a winning region iteratively, the notion of trap is introduced. A trap for the environment (resp. the system) is a region from which the environment (resp. the system) cannot escape without the help of the system (resp. the environment).

Definition 1.2.7. A subset $T \subseteq V$ of the vertices is a *trap* for the environment iff:

- $\forall v \in T \cap V_e, v' \in V,\ (v, v') \in E \implies v' \in T,$
- $\forall v \in T \cap V_s, \exists v' \in T,\ (v, v') \in E$

Traps for system are defined in a similar way.

Using this definition, it is clear that the set $T = \{v_6, v_7\}$ in the arena from Figure 1.2.1 is a trap for the system. From v_6, the system can only go to v_7 and the the environment can chose to move back to v_6. Dually, an attractor for the system to a set $R \subseteq V$ is the set of vertices from which the system can force the game in a finite number of moves to get to a vertex in R. An attractor is constructed as follows.

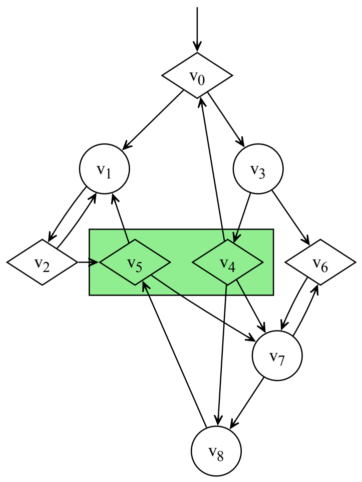
Definition 1.2.8. The system controlled predecessor $\texttt{cpre}(R)$ of $R \subseteq V$ is defined as

$$\begin{aligned} \texttt{cpre}(R) = &\{v \in V_s \mid \exists v' \in R, (v, v') \in E\} \\ &\cup \{v \in V_e \mid \forall v' \in V, (v, v') \in E \implies v' \in R\} \end{aligned}$$

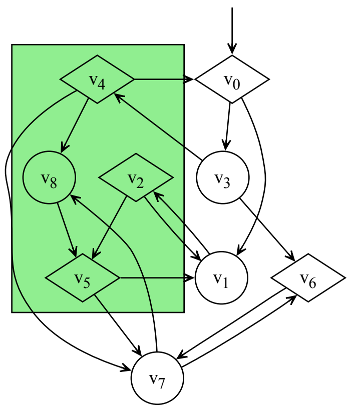
Definition 1.2.9. The system attractor Attr(R) to R is constructed inductively by applying the controlled predecessor:

- $\mathsf{Attr}^0(R) = R$
- $\mathsf{Attr}^{n+1}(R) = \mathsf{Attr}^n(R) \cup \texttt{cpre}(\mathsf{Attr}^n(R))$
- $\mathsf{Attr}(R) = \bigcup_{n \in \mathbb{N}} \mathsf{Attr}^n(R)$

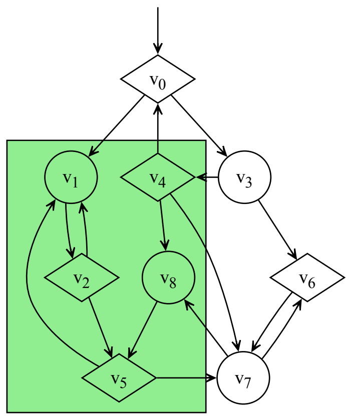
This process is simple enough to be manually applied and it takes at most $|V|$ steps (i.e. applications of the recurrence relation) to converge. Consider the region $R = \{v_4, v_5\}$ from the arena in Figure 1.2.1. Figure 1.2.2 presents the iterative process of constructing Attr(R). In this specific example, a fixed point is reached in 5 steps.



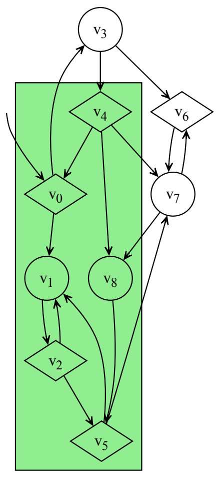
$\mathsf{Attr}^0(R) = \{v_4, v_5\}$



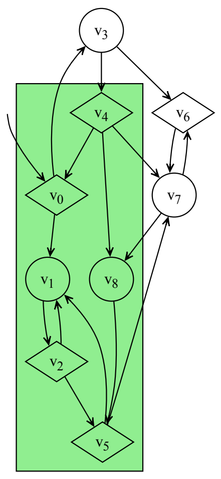
$\mathsf{Attr}^1(R) = \mathsf{Attr}^0(R) \cup \{v_2, v_8\}$



$\mathsf{Attr}^2(R) = \mathsf{Attr}^0(R) \cup \{v_1, v_2, v_4, v_8\}$



$\mathsf{Attr}^3(R) = \mathsf{Attr}^0(R) \cup \{v_0, v_1, v_2, v_4, v_8\}$



$$\mathsf{Attr}^4(R) = \mathsf{Attr}(R) = \{v_0, v_1, v_2, v_4, v_5, v_8\}$$

Figure 1.2.2: Step-by-step construction of Attr(R) with $R = \{v_4, v_5\}$ for the arena in Figure 1.2.1

1.2.2 The winning condition

The base definition of *Win* is generic but solving it is costly. Therefore, we typically resort to more structured winning conditions. Before introducting those, we first need to define two operators:

- For a play $\rho = \rho_0\rho_1\rho_2 \ldots \in V^\omega$, the set of vertices occuring in ρ is defined as

$$\mathsf{Occ}(\rho) = \{v \in V \mid \exists n \in \mathbb{N}, \rho_n = v\}$$

- Similarily, the set of vertices occuring infinitely often in a play ρ is defined as

$$\mathsf{Inf}(p) = \{v \in V \mid \text{ there exists infinitely many } n \in \mathbb{N} \text{ such that } \rho_n = v\}$$

Some games are said to be *dual* to each other. If two games are dual, the description of one of them suffice to deduce the other one. This is useful because we can always reduce a problem to its dual if it makes it easier to solve.

Definition 1.2.10. Let $\mathcal{A} = (V, V_e, V_s, E)$ be an arena. The *dual arena* $\overline{\mathcal{A}}$ of $\mathcal{A}$ is defined as $\mathcal{A} = (V, V_e, V_s, E)$.

In the dual $\mathcal{G}' = (\overline{\mathcal{A}}, \mathsf{Win})$ of a game $\mathcal{G} = (\mathcal{A}, \mathsf{Win})$, every winning strategy for the system σ from a vertex v in $\mathcal{G}$ is a also winning strategy for the environment from v in $\mathcal{G}'$, and conversely.

We can then identify various formalisms to represent the winning condition.

- A *reachability game* is a game in which the system is expected to reach some states that belong to a certain set of vertices. The winning condition of a reachability game induced by a set $R \subseteq V$ is therefore of the form:

  $$\textit{Win} = \{\rho \in V^\omega \mid \text{Occ}(\rho) \cap R \neq \emptyset\}$$

  The dual of a reachability game is a *safety game*. In a safety game, we want the system to *never* leave a set of “safe” vertices $S = V \backslash R$. The winning condition of a safety game is of the form:

  $$\textit{Win} = \{\rho \in V^\omega \mid \text{Occ}(\rho) \subseteq S\}$$

- *Büchi games* can be seen as a generalization of reachability games. A Büchi game is a game in which the system is expected to reach a certain set of vertices infinitely often. The winning condition of a Büchi game induced by a set $F \subseteq V$ is therefore of form:

  $$\textit{Win} = \{\rho \in V^\omega \mid \text{Inf}(\rho) \cap F \neq \emptyset\}$$

  The dual of a Büchi game is called a *co-Büchi game*, which can be seen as the generalization of a safety game. In a co-Büchi game, we want a set of vertices to be visited only *finitely* often by the system. The winning condition of a co-Büchi game induced by a set $C \subseteq V$ is of the form:

  $$\textit{Win} = \{\rho \in V^\omega \mid \text{Inf}(\rho) \subseteq C\}$$

- *Parity games* are a generalization of Büchi games. Each vertex $v \in V$ is assigned a natural number. For the system to win, the smallest number that is seen infinitely often shall be even. In a parity game induced by a mapping $\Omega : V \longrightarrow \mathbb{N}$, a play ρ the winning condition is of the form:

  $$\textit{Win} = \{\rho \in V^\omega \mid \text{min}(\text{Inf}(\Omega(\rho_0), \Omega(\rho_1), \Omega(\rho_2), \ldots)) \text{ is even}\}$$

  It can be noted that since V is finite, $\Omega(V)$ is too so the minimum is always well defined. It is also unique for $\Omega(\rho_0), \Omega(\rho_1), \Omega(\rho_2), \ldots$ (and thus Inf$(\Omega(\rho_0), \Omega(\rho_1), \Omega(\rho_2), \ldots)$) is a subset of $\mathbb{N}$.

1.2.3 Symbolic definition

While the explicit game representation provides an intuitive framework to reason about games, it does so at the expense of succinctness. Transitioning from explicit set notations to a *symbolic* approach allows for a more compact representation of the game. It also addresses the problem of state explosion associated with explicit representations. However, it’s important to note that this may come with the trade-off of a potentially less intuitive formulation of the problem because of the use of μ-calculus as elaborated upon the following pages.

Notations — Primed variables If $\mathcal{V}$ is a set of boolean variables, the set of primed variables $\mathcal{V}' = \{s' | s \in \mathcal{V}\}$ defines the variables after a transition.

Definition 1.2.11. A symbolic game structure $G = (\mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi)$ is defined as follows: **[?]**

- $\mathcal{V} = \mathcal{X} \sqcup \mathcal{Y}$ is a finite set of boolean variables. A state is a subset of $2^{\mathcal{V}}$.
- $\mathcal{X}$ is the set of input variables, that is, those controlled by the environment.
- $\mathcal{Y}$ is the set of output variables, that is, those controlled by the system.
- θ_e is the initial condition of the environment. This is an assertion on $\mathcal{X}$ characterizing the allowed initial states of the environment.
- θ_s is the initial condition of the system. This is an assetion on $\mathcal{Y}$ characterizing the allowed initial states of the system.
- ρ_e is the transition relation of the environment. This is an assertion on $\mathcal{V} \cup \mathcal{X}$. $s_{\mathcal{X}} \in 2^{\mathcal{X}}$ is a *possible* input in state $s \in S^{\mathcal{V}}$ if $(s, s') \models \rho_e$
- ρ_s is the transition relation of the system. This is an assertion on $\mathcal{V} \cup \mathcal{X} \cup \mathcal{Y}$. $s_{\mathcal{Y}} \in 2^{\mathcal{Y}}$ is a *possible* output in state $s \in S^{\mathcal{V}}$ when reading an input $s_{\mathcal{X}} \in 2^{\mathcal{X}}$ if $(s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s$.
- φ is an LTL formula representing the winning condition of the system. From now on, the phrase *winning condition* will refer to this formula.

A state s is said to be *initial* if $s \models \theta_e \wedge \theta_s$. A state s' is the *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. We recognize the main components of the classical game definition. The set $2^{\mathcal{V}}$ corresponds to the vertices of the arena and the edges are modelled by the conjunction of ρ_e and ρ_s. The winning condition is not a set anymore but the LTL formula φ.

The other main concepts defined for classical games are also applicable to symbolic games.

Definition 1.2.12. A *play* is a maximal sequence $\sigma = s_0 s_1 s_2 \ldots$ satisfying *initiality* i.e. s_0 is initial, and *consecution* i.e. for each $i \geq 0$, s_{i+1} is a successor of s_i. σ is said to be winning if $\sigma \models \varphi$

Definition 1.2.13. A *strategy* for the system is a function $f : M \times 2^{\mathcal{V}} \times 2^{\mathcal{X}} \longrightarrow M \times 2^{\mathcal{Y}}$, where M is some memory domain, such that for every $s \in 2^{\mathcal{V}}$, every $s_{\mathcal{X}} \in \mathcal{X}$ and $m \in M$, $f(m, s, s_{\mathcal{X}}) = (m', s_{\mathcal{Y}})$ implies that $(s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s$.

1.3 Modal mu-calculus on games

μ-calculus is a propositional modal logic augmented with fixed points, offering a better expressivity than most time logics such as LTL since it is possible to express them in terms of μ-calculus[6]. It can be defined as an extension of the Boolean logic with the least and greatest fixed point operators, noted respectively μ and ν. *Least* and *greatest* are to understand in the sense of set inclusion. The existence of the two fixed points can be proven using the Knaster-Tarski theorem.

Theorem (Knaster-Tarski Theorem). *Let L be a complete lattice and $f : L \longrightarrow L$ a monotonic application, then the subset of fixed points of f also forms a complete lattice.*

A direct consequence of this theorem is that any monotonic application $f : L \longrightarrow L$ over a complete lattice L admits a least and a greatest fixed-points since a lattice cannot be empty. In particular, the set of states $2^{\mathcal{V}}$ can be viewed as a complete lattice with the order relation $\subseteq$. The Knaster-Tarski theorem thus guarantees the existence of the least and greatest fixed points.

1.3.1 Syntax

The modal μ-calculus over game structures can be constructed as follows, where $v \in \mathcal{V}$ denotes an atomic formula and $Var = \{X, Y, \ldots\}$ a set of relational variable:

$$\varphi ::= v \mid \neg v \mid X \mid \varphi \vee \varphi \mid \text{\textcircled{$\Diamond$}}\varphi \mid \mu Y.\varphi$$

$\neg$ and $\vee$ have the same meaning as in temporal logic and $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined in them same way. We'll see in next section that $\text{\textcircled{$\Diamond$}}$ is analogous to the controlled prececessor `cpre` defined in Section 1.2.1 and μ is the "least fixed point" operator. The "greatest fixed point" operator ν is defined as the dual of μ, i.e. $\nu Y.\varphi(Y)$ abbreviates $\neg\mu Y.\varphi(\neg Y)$.

Important note When writing $\mu Y.\varphi(Y)$, it is implied that φ is syntactically monotonic in the propositional variable Y. This means that all occurences of Y in φ are preceded by an even number of negations.

1.3.2 Semantics

In μ-calculus over a game structure G, a formula φ is interpreted as the set of states in which φ is true. Such a set is noted $[[\varphi]]_G$ where G is the game structure, or simply $[[\varphi]]$ when G is clear from context. Let φ be a formula and let $s \in 2^{\mathcal{V}}$, then $\varphi[X/s]$ denotes the formula φ in which all occurences of the relational variable X have been replaced by s. For a finite games structure G, $[[\varphi]]_G$ is thus defined inductively as follows:

- $[[v]] = \{s \in 2^{\mathcal{V}} \mid v \in s\}$
- $[[\neg v]] = \{s \in 2^{\mathcal{V}} \mid v \notin s\}$

- $[[X]] = \emptyset$[2]
- $[[\varphi \vee \psi]] = [[\varphi]] \cup [[\psi]]$
- A state s is in $[[\diamondsuit \varphi]]$ if, regardless of how the environment moves from s, the system can chose an appropriate move to reach a state in $[[\varphi]]$, that is:

$$[[\diamondsuit \varphi]] = \left\{ s \in 2^{\mathcal{V}} \middle| \begin{array}{l} \forall s_{\mathcal{X}} \in 2^{\mathcal{X}}, (s, s_{\mathcal{X}}) \models \rho_e \longrightarrow (\exists s_{\mathcal{Y}} \in \mathcal{Y}, \\ (s, s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \rho_s \wedge (s_{\mathcal{X}}, s_{\mathcal{Y}}) \models \varphi) \end{array} \right\}$$

  It can be noted that $\varphi \longrightarrow \diamondsuit \varphi$ is monotonic which implies the existence of fixed-points.

- $[[\mu X.\varphi]] = \bigcup_{n \in \mathbb{N}} S_n$ where $S_0 = \emptyset$ and $S_{n+1} = [[\varphi[X/S_n]]]$

Another possible definition for the least fixed-point operator is a recursive one. As opposed to the abstract one presented above, it might provide a more intuitive picture. Let φ be a monotonic formula in the variable X, then:

$$\mu X.\varphi \equiv \varphi[X/\mu X.\varphi]$$

This means that reasoning about $\mu X.\varphi$ is equivalent to reasoning about $\varphi[X/\mu X.\varphi]$. Hence, we can easily provide a definition for the attractor set similar to the one presented in Definition 1.2.9:

Definition 1.3.1. On a symbolic game structure $G = (\mathcal{V}, \mathcal{X}, \mathcal{Y}, \theta_e, \theta_s, \rho_e, \rho_s, \varphi)$, the system attractor to a set of states $S \subseteq 2^V$ is defined as

$$Attr(S) = \mu X.(S \vee \diamondsuit X)$$

This definition uses the least fixed point because the greatest fixed point is simply $2^{\mathcal{V}}$.

1.4 Generalized Reactive(1) synthesis

One classical approach for the synthesis problem is to start from a specification with input variables controlled by the environment and output variables controlled by the system. This specification is then translated to a two-player game in which the goal is to find a winning strategy for the system. Finding such a winning strategy is equivalent to finding a controller. Hower, this can only be done in doubly-exponential time [10].

As said in the previous pages, we consider a fragment of LTL called "Generalized Reactive(1)" which allows to perform synthesis in polynomial time. GR(1) formulas are an implication between a conjunction of assumptions and a conjunction of guarantees of the form:

[2]We are only interested in cases were all variables are bound by either μ or ν, but if we wanted to allow free variable, we would have to consider the initial state of the environment

$$\left(\theta_e \wedge \square\rho_e \wedge \bigwedge_i \square\lozenge a_i\right) \longrightarrow \left(\theta_s \wedge \square\rho_s \wedge \bigwedge_j \square\lozenge g_j\right) \tag{1.4.1}$$

where:

- θ_e and θ_s are the initial conditions of the environment and the system
- ρ_e and ρ_s are the transitions of the systems and environment, also called *safety* properties
- a_i and g_j are the the i^{th} and j^{th} *liveness* properties of the system and environment, that is, statements that need to be verified infinitely often.

Let us unfold Equation 1.4.1. We are interested in open systems, i.e. systems that interact with an uncontrolled environment that might be adverserial. Hence, it seems reasonable to have *assumptions* about the environment describing the things we assume will happen infinitely often. On the other hand, we want the system to perform some actions infinitely often. *If* we assume that environment behaves correctly, we guarantee that the system will meet its requirements. As for the safety properties, they constraint the system and the environment in their current choices.

There are two ways for the system to win:

1. The system manages to attract the environment in a trap where it cannot fullfill all its assumptions.
2. The system manages to always fullfill its guarantees.

Let $\{a_1, \ldots, a_n\}$ be a set of n liveness assumptions and $\{g_1, \ldots, g_m\}$ a set of m liveness guarantees. This translates to the following winning condition [5]:

$$\varphi_{\mathsf{GR}} = \nu Z. \bigwedge_{i=1}^{n} \mu Y. \bigvee_{j=1}^{m} \nu X. \circledcirc((g_j \wedge \bigcirc Z) \vee \bigcirc Y \vee (\neg a_i \wedge \bigcirc X)) \tag{1.4.2}$$

Formula 1.4.2 can be hard to understand at first sight so let us annotate it with colors:

$$\varphi_{\mathsf{GR}} = \nu\mathbf{Z}. \bigwedge_{i=1}^{n} \mu\mathbf{Y}. \bigvee_{j=1}^{m} \nu\mathbf{X}. \circledcirc((\mathbf{g_j} \wedge \bigcirc \mathbf{Z}) \vee \bigcirc \mathbf{Y} \vee (\neg\mathbf{a_i} \wedge \bigcirc \mathbf{X}))$$

- The green part corresponds to the system fullfilling its guarantees. It is associated with a greatest fixed-point operator since this must be true *infinitely often* to hold and because we want the winning region to be as big as possible.
- The pink part corresponds to the environment *not* fullfilling its assumptions and thus violating the left-hand side of the implication from Formula 1.4.1. This is also incuded in the winning region, that we want as big as possible, which explains the greatest fixed point.

- The blue part corresponds to the *unconstrained* part of a play. It is associated to a least fixed-point since this phase must be finite.

Chapter 2

Motion planning

Motion planning is the problem of navigating a controlled system from a source to a destination while satisfying real-world physical constraints. What makes this problem even more involved is that the motions of the body can be governed by complex dynamics of the form

$$\ddot{q} = f(q, \dot{q}, u)$$

where q is a vector representing the state of the system, u is the input vector and f is non-linear. In such cases, exact methods are often computationally intractable, if even possible. Fortunately, terminating algorithms exist to approximate the actual solution. Specifically, we are interested in *sampling-based* planning. Section 2.1 presents general concepts of motion planning while Section 2.2 describes some popular sampling-based motion algorithms.

2.1 The configuration space

The task to be performed is represented in a workspace $\mathcal{W}$, which is usually $\mathbb{R}^2$ or $\mathbb{R}^3$, as it only takes into account the spatial coordinates. The configuration space $\mathcal{C}$, or C-space, however, is the set of all possible configurations of the robot. Each configuration is represented as a unique point in an space whose dimension depends on the system. In the case of cycle-free mechanisms, the size of the C-space is the number of degrees of freedom of the robot. This abstraction allows to represent various problems with very different constraints with the same algorithms.

2.1.1 Degrees of freedom

Robots are commonly represented as a collection of connected rigid bodies. To determine the number of degrees of freedom of the robot, we first need to determine the number of degrees of freedom of a rigid body. To fully describe the configuration of a rigid body in $\mathbb{R}^2$, we need three coordinates, two of which are linear, x and y, and one of which is its angle θ, sometimes called yaw. To get the total number of degrees of freedom of a cycle-free mechanism, we sum the degrees of freedom of each of its rigid body and we substract the number of independent constraints.

To understand this formula, let us consider the two-joints planar robotic arm of Figure 2.1.1. It is composed of two 2-dimensional links that would each have 3 degrees of freedom if they were unconstrained. Each joint applies constraints on both the x and y-coordinates of the links so the total number of constraints is 4. We assume their independence. The total number of degrees of freedom for this robotic arm is thus $6 - 4 = 2$. This means that its configuration space is 2-dimensional.

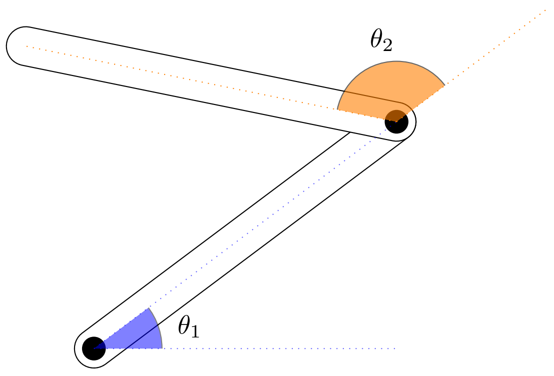


Figure 2.1.1: Two-joints planner robotic arm

2.1.2 C-space obstacles

Any C-space $\mathcal{C}$ can be partitioned into two subsets: $\mathcal{C}_{obs}$, the set of configurations for which the system is in collision with an obstacle or itself and $\mathcal{C}_{free}$, which is simply $\mathcal{C} \setminus \mathcal{C}_{obs}$. More formally, if we ignore self-collisions, $\mathcal{C}_{obs}$ can be expressed as

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}$$

where $\mathcal{W}$ is the workspace, $\mathcal{O} \subset \mathcal{W}$ is the obstacle region, $\mathcal{A}$ is the geometrical model of robot and $\mathcal{A}(q) \subset \mathcal{W}$ denotes the position of the robot on configuration q, then. Once $\mathcal{C}_{obs}$ is constructed, the basic motion planning is much simpler for it simply consists of finding a path in C_{free}.

Let us again consider the robotic arm from Figure 2.1.1 in a workspace containing to rectangular obstacles as pictured in Figure 2.1.2. A representation of its C-space obstacles is given in Figure 2.1.3. In this example, since $\mathcal{C}_{free}$ is a connected region, the robotic arm can move freely in $\mathcal{W} \setminus \mathcal{O}$ from any start configuration.

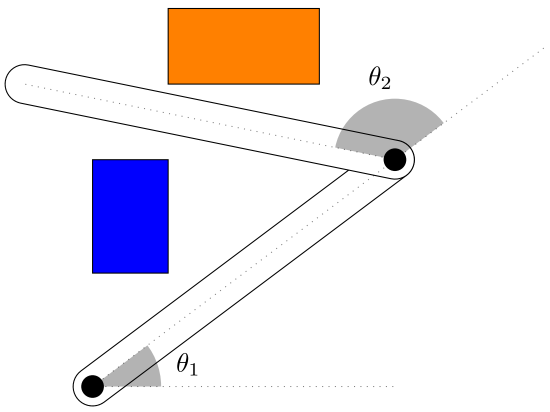


Figure 2.1.2: Two-joints planner robotic arm with two rectangular obstacles

Note Both degrees of freedom are angles and since $2\pi = 0 \pmod{2\pi}$, the topology of the C-space is a sphere so the top and bottom edges, as well as the left and right edges, are connected.

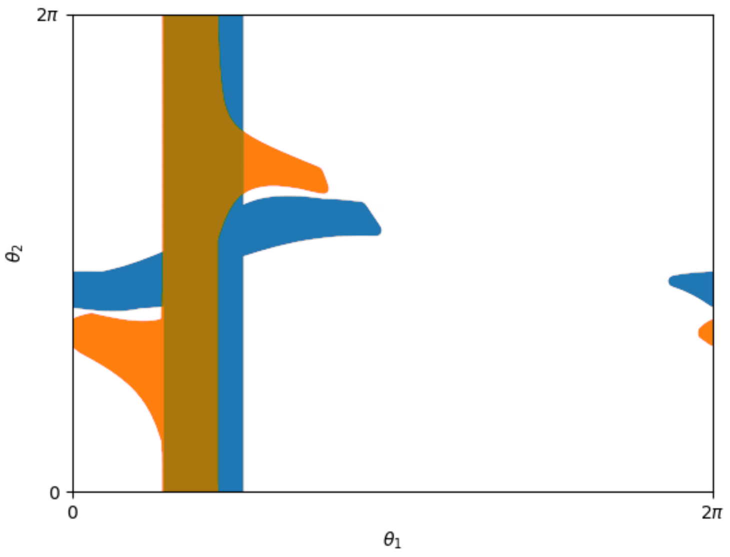


Figure 2.1.3: C-space obstacles representation associated to Figure 2.1.2

2.2 Sampling-based methods

Unfortunately, most problems yield complicated C-space obstacles, which makes the exact determination of $\mathcal{C}_{obs}$ inconceivable. Moreover, the geometric model of the robot is typically not a bijection, which means that $\mathcal{A}^{-1}$ may not have an analytical form. Sampling-based motion planning addresses this issue by never explicitely constructing the C-space obstacles. Some of the most used sampling-based motion planning algorithms are Probabilistic roadmaps (PRM) [7] and Rapidly-exploring random trees (RRT) [9]. Both have been used to solve complex motion planning problems in high-dimensional C-spaces and both are probabilistically complete under relatively weak conditions.

Definition 2.2.1. A *complete* motion planner is a motion planner that will always find a solution if one exists or else concludes that no path exists.

Definition 2.2.2. A *probabilistically complete* motion planner is a motion planner that finds a solution if one exists with a probability of 1 when the run time goes to infinity.

Note from this definition that a planner that is only probabilistically complete cannot conclude in a finite time when no path exists.

2.2.1 Probabilistic roadmaps

PRMs are a way to approximate the free space by randomly sampling points and creating connections between them. The basic algorithm can be outlined as follows:

Algorithm 1 PRM algorithm — Construction phase

```
Output: The roadmap $R = (N, E)$

$V \leftarrow \emptyset$
$E \leftarrow \emptyset$
loop
    $q \leftarrow$ a randomly chosen free configuration
    $N_q \leftarrow$ a set of candidate neighbors of $q$ chosen from $V$
    $V \leftarrow V \cup \{q\}$
    for all $q' \in N_q$ do
        if $(q, q') \notin E$ and TRY_FIND_PATH$(q, q')$ then
            $E \leftarrow E \cup \{(q, q')\}$
        end if
    end for
end loop
```

where TRY_FIND_PATH(q, q') returns true iff the local planner is able to find a path from q to q', or at least a state close to q'

After this construction phase, a roadmap has been made and an optimal path can be found with a best-first graph searching algorithm such as Djikstra's algorithm or A^*. Since the learning phase of PRM can take a long time, it is generally used in cased where multiple queries are required on a static configuration space.

2.2.2 Rapidly exploring random trees

RRTs are designed to search a high-dimensional configuration space by building a space-filling tree. The incremental construction of RRTs has interesting properties such as not needing to point-to-point local planner or beeing biased towards unexplored regions. The basic algorithm is outlined follows:

Algorithm 2 RRT algorithm

```
Input: $x_{init} \in C_{free}$ an initial configuration; $K$, the desired size of the tree; $\Delta t$ a time interval
Output: An RRT $\mathcal{T}$

  $\mathcal{T} \leftarrow$ a tree with a single vertex $q_{init}$
  for $k \leftarrow 1 to K$ do
      $q_{rand} \leftarrow$ a randomly chosen free configuration
      $q_{near} \leftarrow$ the nearest neighbor of $q_{rand}$ in $\mathcal{T}$
      $u_{new} \leftarrow \texttt{SELECT\_INPUT}(q_{near}, q_{rand})$
      $q_{new} \leftarrow$ apply $u$ from $x_{near}$ $\Delta t$
      Add the vertex $q_{new}$ to $\mathcal{T}$
      Add the edge $(q_{near}, q_{new}, u_{new})$ to $\mathcal{T}$
  end for
```

where SELECT_INPUT(q_{near}, q_{rand}) computes an input that allows to move from q_{near} towards the direction of q_{rand}.

RRTs are designed to cover the space as fast as possible and are therefore relatively quick to compute while the algorithm is easy to implement and customize. Contrary to PRMs, RRTs are mostly used for single queries or multiple queries in dynamic environments.

Chapter 3

Combining reactive synthesis and motion planning

We want to control a robot with complex dynamics for an infinite-duration execution in a reactive way. The first difficulty that naturally arises from this description of the problem is that the robot is supposed to run for an infinite time. However, it is impossible to plan an infinite-length path. Hence, we need to create separate paths that will need to be plugged together. For that, we assume the existence some set of *sandboxes* that will be described in Section 3.1.2. Those sandboxes allow the planned paths to be connected so that they all form a closed path, making it possible to loop infinitely often on them.

Another concern is that the synthesis process is *discrete*: the building blocks of an LTL formula are atomic propositional variables and the output of the synthesis is a discrete controller. The real world, however, is continuous. It is therefore necessary to implement (i) a way to express a real-world task using propositional variables (ii) a representation of motions that is abstract enough to be understood by both the synthesis tool and the motion planner. Section 3.1.1 provides a solution for the first issue and the structure used to address the second one is presented in Section 3.2.

3.1 The workspace

The workspace in which the robot evolves is composed of three fixed parts: a set of obstacles, a set of regions of interest, and a set of sandboxes. The set $\mathcal{O}$ of obstacles doesn't need to be described in this section since it is a general feature of all motion planning problems, described in Chapter 2. Figure 3.1.1 presents such a workspace with an example solution.

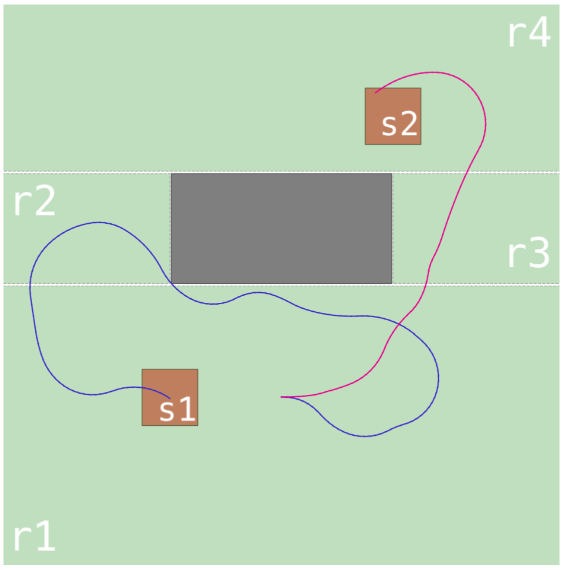


Figure 3.1.1: Workspace with a motion planning tree, four regions, two sandboxes and an obstacle.

3.1.1 Regions of intereset

The workspace $\mathcal{W}$ contains a set $\mathcal{R}$ of regions of interest. We assume those regions to be distinct and, for each $R \in \mathcal{R}$, we assume that $R \cap \mathcal{O} = \emptyset$. From a high level, we are only interested in the motion between those regions while the motions inside each region can be arbitrary. We associate an atomic proposition to each region of interest and, by abuse of notation, we note it R_i as well. A possible example could be:

If some input bit is set, patrol a set $\{R_1, \ldots, R_k\} \subseteq R$ of regions infinitely often.

which corresponds to the following LTL formula:

$$b \longrightarrow \bigwedge_{i=1}^{k} \Box \Diamond R_i$$

Note that this is a GR(1) specification with $\theta_e = b$ and assumptions being trivially true.

Each region is a simple polygon for which we provide the motion planner with a geometric description. We also define an adjancecy relation $\tau \subseteq R \times R$. $(R_a, R_b) \in \tau$ means that the robot can possibly be controlled to go from R_a to R_b. This adjacency relation will be useful for the process described in Section 3.3.

3.1.2 Sandboxes

To plug the planned paths together and enable the system to follow loops, we define a set of sandboxes $\mathcal{S}$.

Definition 3.1.1. Let $\mathcal{C}$ be the configuration space associated to the workspace $\mathcal{W}$. A *sandbox* in a region of interest R is a couple $S = (E, p)$ with $E \subseteq R$ and $p \in \mathcal{C}$ such that for every $\boldsymbol{x_0} \in E$, there is a control law κ_S that brings the robot from position $\boldsymbol{x_0}$ to the state p. p will be referred as the *output state* of the sandbox, and κ_S is called the *sandbox controller* associated with S.

Each sandbox is in a region of interest but its output state doesn't have to. However, the sandbox controller must output paths that are collision-free. Once the set of sandboxes is computed, it is sufficient to state that each planned path must start and end in a sandbox to make sure that they are connected.

3.2 Abstract trees

The planned paths will be the output of the motion planner, but also part of the input of the synthethis process. It is therefore necessary to give an abstract definition that is not dependant on the exact geometry of the real world.

We use a sampling-based tree planner such as RRT to generate paths. Two important properties have been stated in Section 3.1 and we will use them to define the trees produced by the motion planner from a high level:

- Each motion planning tree must start and end in a sandbox. This means that the root and leaves of each motion planning tree must be sandboxes.
- The high-level robot motions are only represented by the inter-region moves. The nodes of the motion planning trees that are not the root nor leaves must, therefore, represent region changes.

The workspace in Figure 3.1.1 has a single motion planning tree, which is divided in two branches. Figure 3.2.1 depicts the abstract tree that represents this motion planning tree.

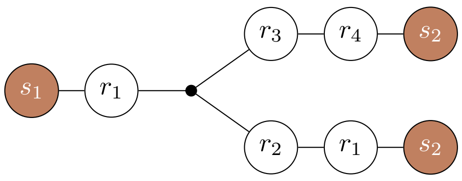


Figure 3.2.1: Abstract motion planning T_1 tree summarizing Figure 3.1.1.

Choice of the motion planning tree The question arises as to which tree the motion planning algorithm should aim for when searching for a tree. For instance, the tree T_2 depicted in Figure 3.2.2 also corresponds to the tree from Figure 3.1.1.

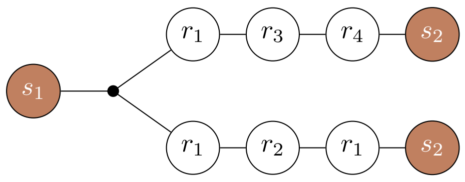


Figure 3.2.2: Abstract motion planning T_2 simulated by T_1.

T_2 has more nodes than T_1 while not providing more information. To chose a tree that is as general as possible up to depth k, we use the following criterion:

> Given some motion tree depth k, we want to find a motion planning tree that simulates all motion planning trees with depth at most k.

Definition 3.2.1. A tree *simulates* another tree if all behaviors on the other tree are contained on the simulating tree. Additionally, the paths on the simulating tree must contain at least the paths of the simulated trees and at the same time no decision must be made in the simulating tree until a corresponding decision is made in the simulated tree

This definition of simulation ensures that for moderate values of k, the generated motion planning trees do not have to become prohibitively large.

3.3 Abstraction-refinment loop

We maintain a set of abstract motion planning trees $\mathcal{T}$ which is extended at runtime. New trees are automatically derived by the synthesis procedure. Those are then used as input for the motion planner. If, for a given edge of a motion planning tree, it could not find a solution, it shall only return the parts of the tree that could be planned and add them to $\mathcal{T}$.

Each transition in an abstract motion planning tree is associated with a cost. Whenever a transition cannot be planned with the given resources, its cost is increased. The synthesis procedure seeks to minimize the cost of its solution, therefore penalizing those that are harder to plan. It ensures convergence since it won't be able to generate the same candidates over and over again. On the other hand, the resources allotted to the motion planner to find a given path are proportional to the cost of the associated transition. In practice, this means that a path that is deemed harder to find will be given more time.

The cost function $Cost_{\mathcal{T}} : \mathcal{T} \times \tau \longrightarrow \mathbb{R}$ represents the difficulty of finding a transition in τ from a state in a motion planning tree. Since the synthesis procedure can also generate entirely new motion planning trees, we also need a cost function $Cost_R$ to represent the cost of a transition to such a tree.

A simplified version of the full algorithm is given by Algorithm 3.

Algorithm 3 Simplified abstraction-refinement loop

```
1: $\mathcal{T} \leftarrow \emptyset$
2: loop
3:     Synthesize a controller $C$ using $\mathcal{T}$, $Cost_{\mathcal{T}}$ and $Cost_R$
4:     $\mathcal{T}' \leftarrow$ motion planning trees used in $C$
5:     if $\mathcal{T}' = \mathcal{T}$ then
6:         return $\mathcal{T}$
7:     end if
8:     for $t \in \mathcal{T} \cap \mathcal{T}'$ do
9:         if $t$ can be fully planned then
10:            Add $t$ to $\mathcal{T}$
11:        else
12:            Increase the cost of the first unplanned transitions in $Cost_{\mathcal{T}}(t, \ldots)$
13:        end if
14:    end for
15:    for $t \in \mathcal{T}' \setminus \mathcal{T}$ do
16:        if $t$ can be fully planned then
17:            Add $t$ to $\mathcal{T}$
18:        else
19:            Increase the cost of the first unplanned transitions in $Cost_R(t, \ldots)$
20:        end if
21:    end for
22:    Remove all trees that are simulated by another tree in $\mathcal{T}$
23: end loop
```

3.3.1 Writing the specification

The specification describes a state machine in which each state corresponds either to:

1. A sandbox
2. A state in a known motion planning tree
3. A state in a tree that is yet to be planned.

As $\mathcal{T}$ is initially empty, the only transitions allowed at the beginning are directly derived from the adjacency relation τ. The only possible states are, therefore, those from the first and third kind. As costs are being modified by the motion planner, the state machine is updated. When a solution is found, all states correspond to either a sandbox or a state in a planned tree.

3.3.2 Finding a path between two regions

To find a path between two regions, the motion planner is called with the first region as a starting point and the second one as a destination. It is however important to consider what happens in between.

If the specification contains a safety property that negates a region, such as $\square \neg R_1$, then the region R_1 can never be crossed. This means that any path between two regions must avoid this region. A solution could be to consider each region except the starting and end region as obstacles for the motion planner. This comes with the caveat of limiting the capabilities of the motion planner, potentially increasing the time required to find a solution.

Another approach is to only avoid regions that need to be avoided, that is, regions that are negated in a safety property. This is done by converting safety properties to negation normal form. Algorithm 4 details this procedure.

Algorithm 4 Extraction of the negated safety variables

```
1: $\mathcal{N} \leftarrow \emptyset$
2: $\mathcal{S} \leftarrow$ safety properties
3: for $\varphi \in \mathcal{S}$ do
4:     $\varphi_{NNF} \leftarrow$ negation normal form of $\varphi$
5:     $\mathcal{N}_\varphi \leftarrow$ negated variables in $\varphi_{NNF}$
6:     $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{N}_\varphi$
7: end for
8: return $\mathcal{N}$
```

Negation normal form (NNF) In NNF, the only allowed operators are $\wedge$, $\vee$ and $\neg$. More importantly, $\neg$ can only be applied to variables. Consider the following formula:

$$\varphi = P \vee (Q \longrightarrow R) \vee \neg(\neg P \wedge Q)$$

This translates in NNF to:

$$\varphi = P \vee (\neg Q \vee R) \vee (P \vee \neg Q)$$

Which simplifies as:

$$\varphi = P \vee \neg Q \vee R$$

The only negated variable in φ is therefore Q.

Bibliography

[1] Rajeev Alur and Salvatore La Torre. “Deterministic generators and games for LTL fragments”. In: *ACM Transactions on Computational Logic (TOCL)* 5.1 (2004), pp. 1–25.

[2] R. Bloem et al. “Synthesis of reactive(1) designs.” In: *Journal of Computer and System Sciences* (2011).

[3] Alonzo Church. “Logic, arithmetic and automata”. In: *Proceedings of the international congress of mathematicians*. Vol. 1962. 1962, pp. 23–35.

[4] Alexandre Duret-Lutz et al. “From Spot 2.0 to Spot 2.10: Whats New?” In: *International Conference on Computer Aided Verification*. Springer. 2022, pp. 174–187.

[5] R. Ehlers and V. Raman. “SLUGS: Extensible GR(1) synthesis”. In: July 2016, pp. 333–339.

[6] E. Allen Emerson. “Model Checking and the Mu-calculus”. In: *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop 1996, Princeton, New Jersey, USA, January 14-17, 1996*. Ed. by Neil Immerman and Phokion G. Kolaitis. Vol. 31. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1996, pp. 185–214.

[7] L.E. Kavraki et al. “Probabilistic roadmaps for path planning in high-dimensional configuration spaces”. In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[8] Hadas Kress-Gazit, Morteza Lahijanian, and Vasumathi Raman. “Synthesis for Robots: Guarantees and Feedback for Robot Behavior”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 1.1 (2018), pp. 211–236.

[9] Steven M. LaValle. “Rapidly-exploring random trees : a new tool for path planning”. In: *The annual research report* (1998).

[10] A. Pnueli and Roni Rosner. “On the synthesis of a reactive module”. In: *Automata Languages and Programming* 372 (Jan. 1989), pp. 179–190.

[11] M. Zimmerman, F. Klein, and A. Weinert. “Lecture notes on Infinite Games”. In: (2016).