LRE
LABORATOIRE DE RECHERCHE DE L'EPITA

Solving 2-Player Games

Specialized data structure for Büchi game solving

Quentin Rataud, under the supervision of Philipp Schlehuber-Caissier

Seminar — July 2024

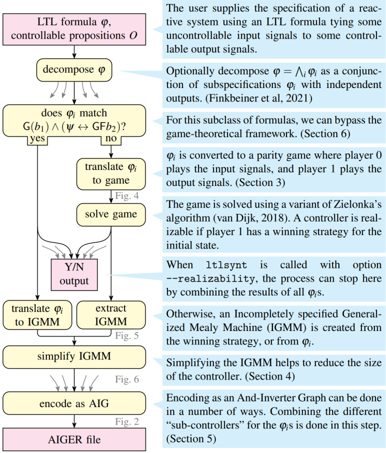


Figure: General outline of the process used by `ltlsynt` to solve the reactive synthesis problem. [2]

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)
Output The traffic light can either turn green (G) or turn red (R)

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)
Output The traffic light can either turn green (G) or turn red (R)

- If no car is detected, the traffic light must be red

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)

Output The traffic light can either turn green (G) or turn red (R)

- If no car is detected, the traffic light must be red
- The traffic light cannot be green twice in a row

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)
Output The traffic light can either turn green (G) or turn red (R)

- If no car is detected, the traffic light must be red
- The traffic light cannot be green twice in a row
- If a car is detected, the traffic light must eventually turn green

A smart traffic light

`ltlsynt` outline

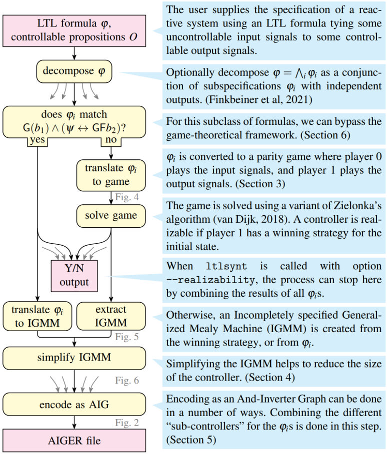


Figure: General outline of the process used by `ltlsynt` to solve the reactive synthesis problem. [2]

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)

Output The traffic light can either turn green (G) or turn red (R)

- If no car is detected, the traffic light must be red
- The traffic light cannot be green twice in a row
- If a car is detected, the traffic light must eventually turn green

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)
Output The traffic light can either turn green (G) or turn red (R)

- $G(!C \implies R)$
- The traffic light cannot be green twice in a row
- If a car is detected, the traffic light must eventually turn green

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)
Output The traffic light can either turn green (G) or turn red (R)

- $G(!C \implies R)$
- $G(G \implies X(R))$
- If a car is detected, the traffic light must eventually turn green

Input A sensor detects if a car is waiting right next to the traffic light (C) or not (!C)
Output The traffic light can either turn green (G) or turn red (R)

- $G(!C \implies R)$
- $G(G \implies X(R))$
- $G(C \implies F(G))$

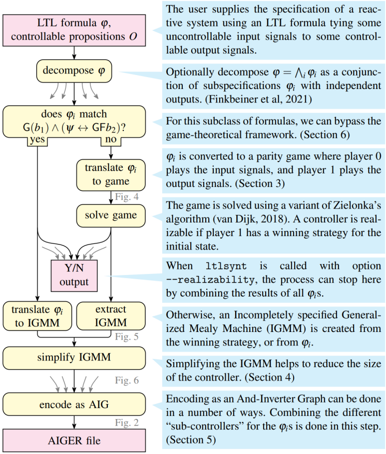


Figure: General outline of the process used by `ltlsynt` to solve the reactive synthesis problem. [2]

A smart traffic light

Turning LTL into a 2-Player Game

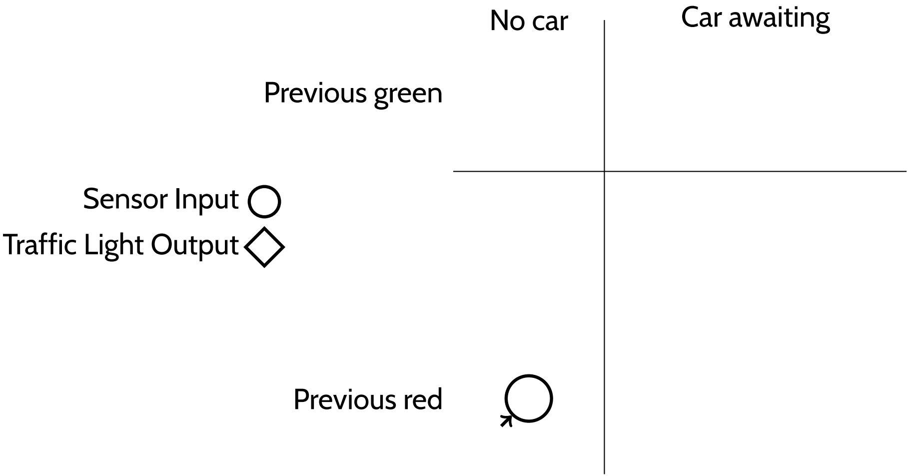


Figure: $G(!C \implies R)\ \&\ G(G \implies X(R))\ \&\ G(C \implies F(G))$

A smart traffic light

Turning LTL into a 2-Player Game

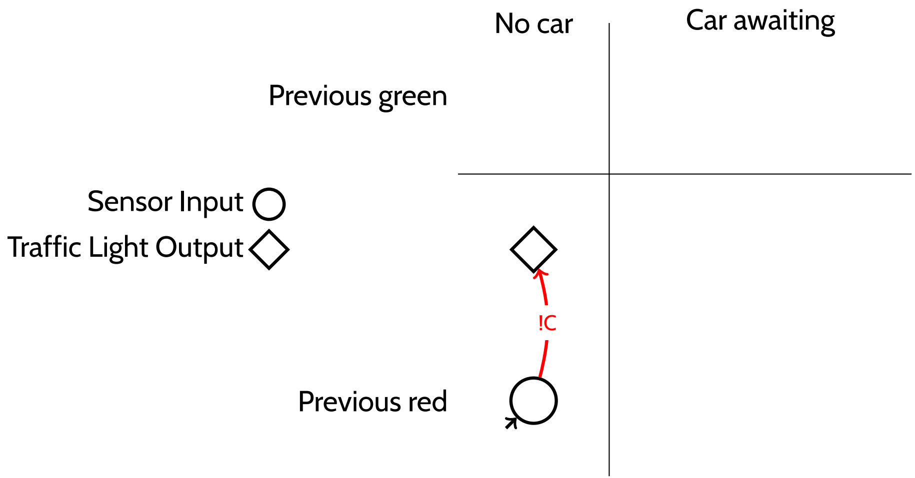


Figure: $G(!C \implies R) \,\&\, G(G \implies X(R)) \,\&\, G(C \implies F(G))$

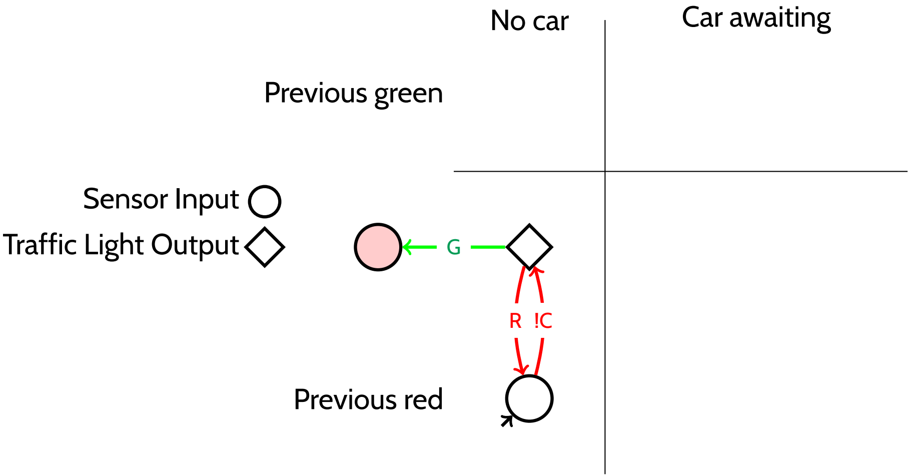


Figure: $G(!C \implies R)\ \&\ G(G \implies X(R))\ \&\ G(C \implies F(G))$

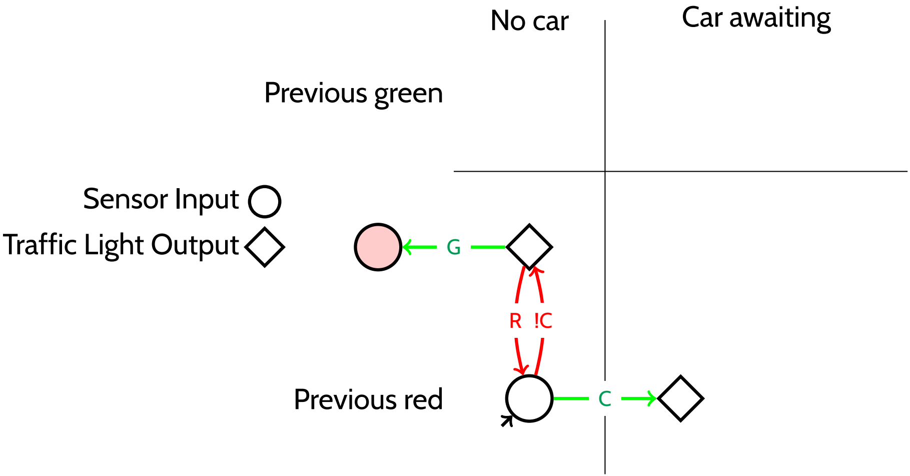


Figure: $G(!C \implies R)$ & $G(G \implies X(R))$ & $G(C \implies F(G))$

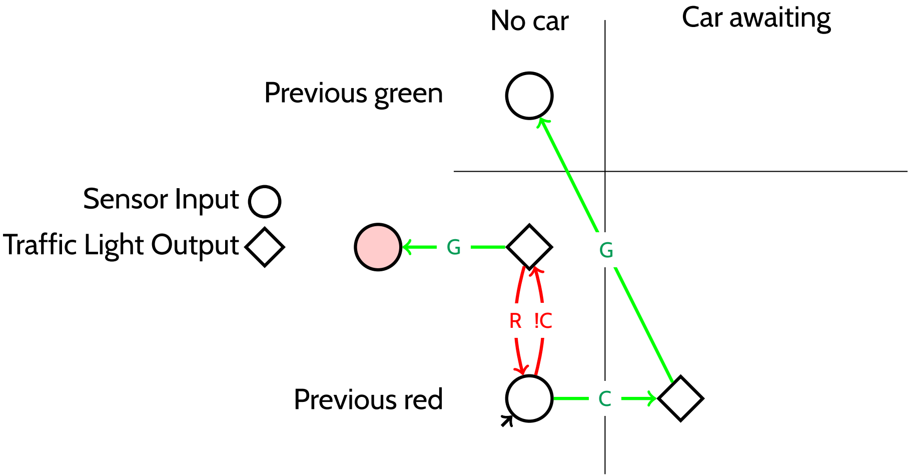


Figure: $G(!C \implies R)\ \&\ G(G \implies X(R))\ \&\ G(C \implies F(G))$

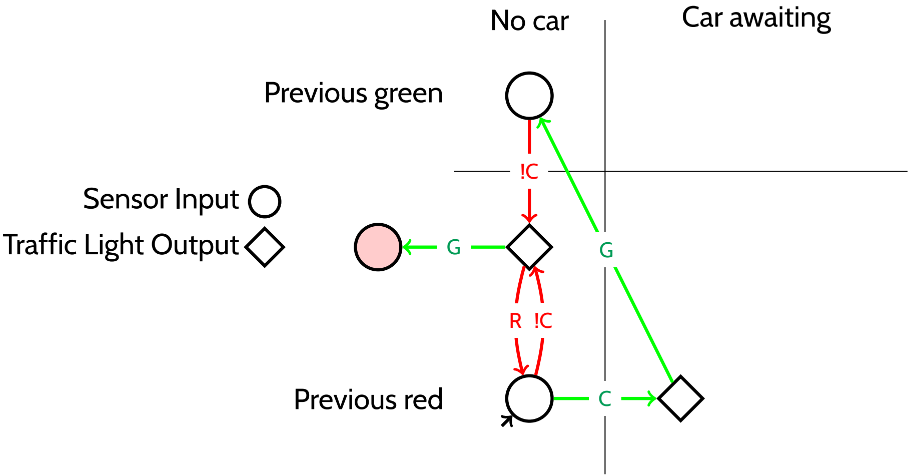


Figure: $G(!C \implies R)$ & $G(G \implies X(R))$ & $G(C \implies F(G))$

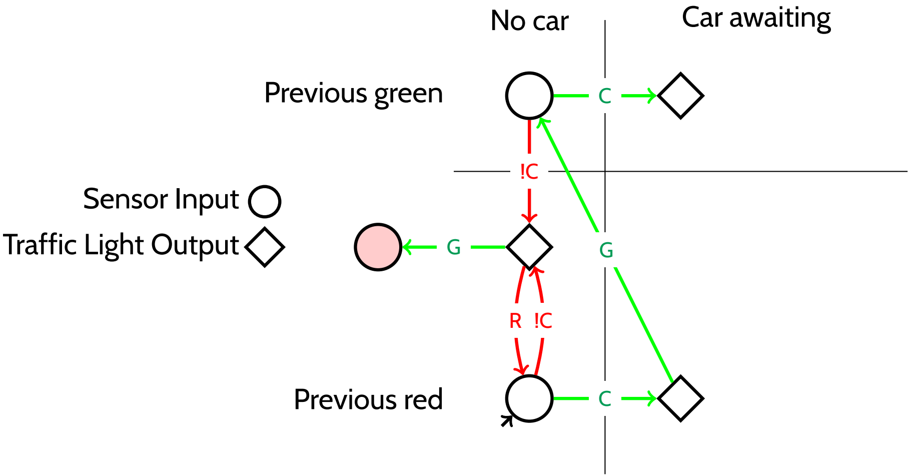


Figure: $G(!C \implies R)$ & $G(G \implies X(R))$ & $G(C \implies F(G))$

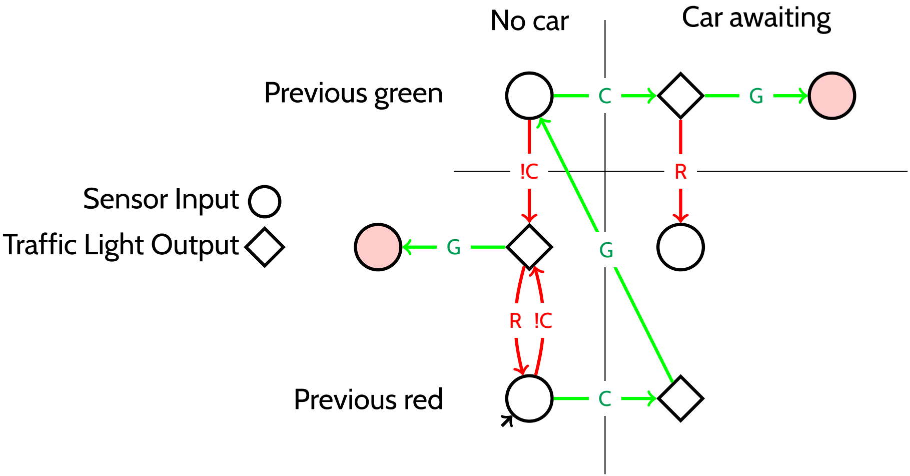


Figure: $G(!C \implies R)$ & $G(G \implies X(R))$ & $G(C \implies F(G))$

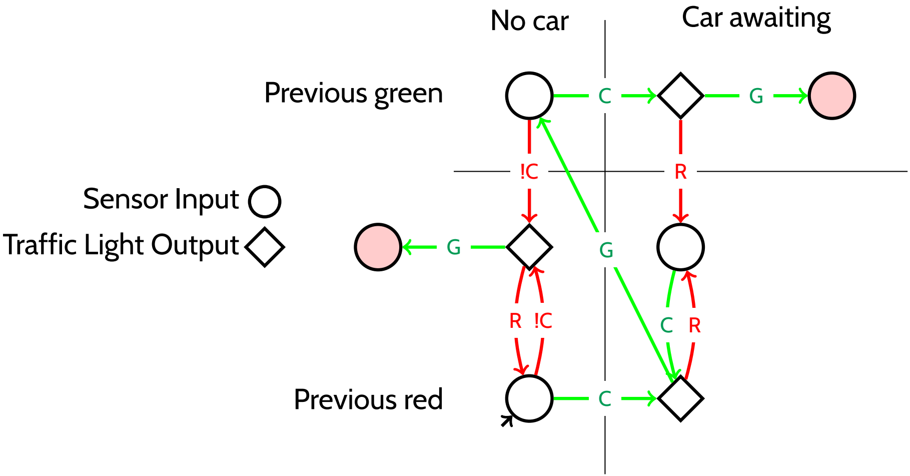


Figure: $G(!C \implies R) \,\&\, G(G \implies X(R)) \,\&\, G(C \implies F(G))$

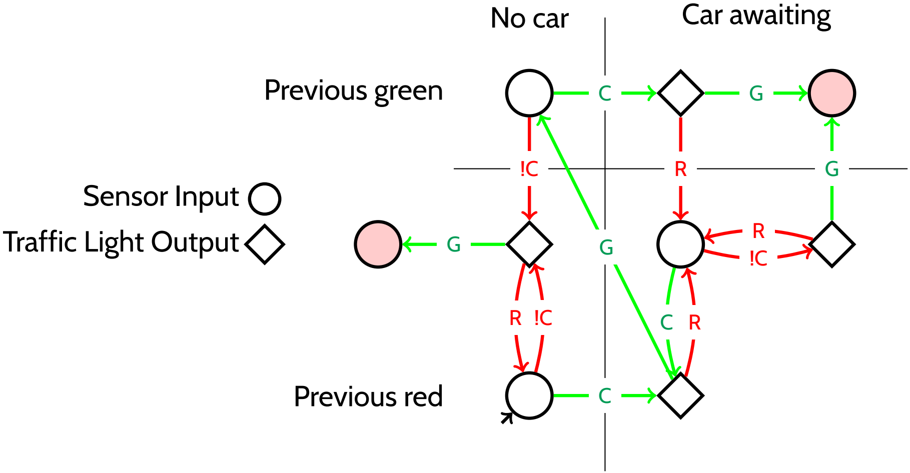


Figure: $G(!C \implies R)$ & $G(G \implies X(R))$ & $G(C \implies F(G))$

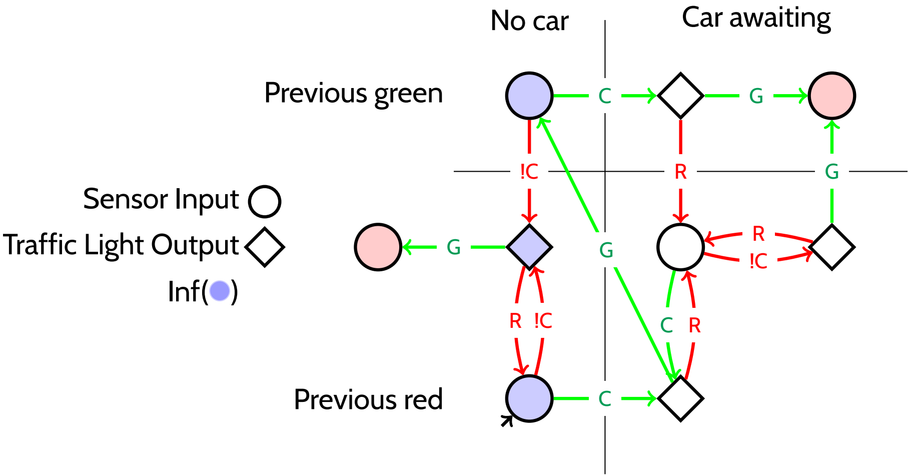


Figure: $G(!C \implies R)\ \&\ G(G \implies X(R))\ \&\ G(C \implies F(G))$

A smart traffic light

`ltlsynt` outline

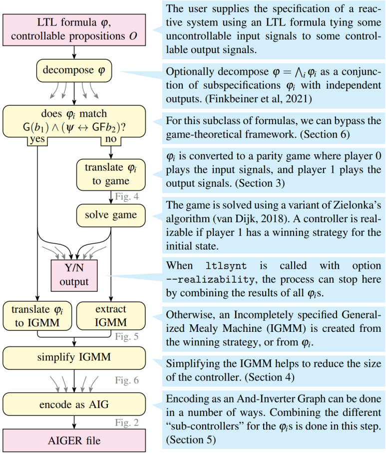


Figure: General outline of the process used by `ltlsynt` to solve the reactive synthesis problem. [2]

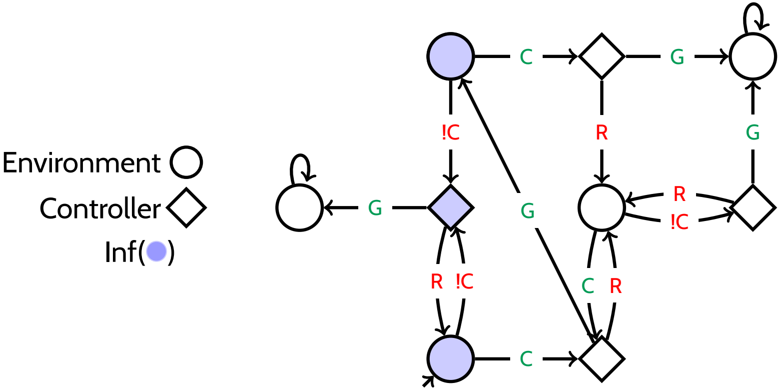


Figure: Solving a Büchi game

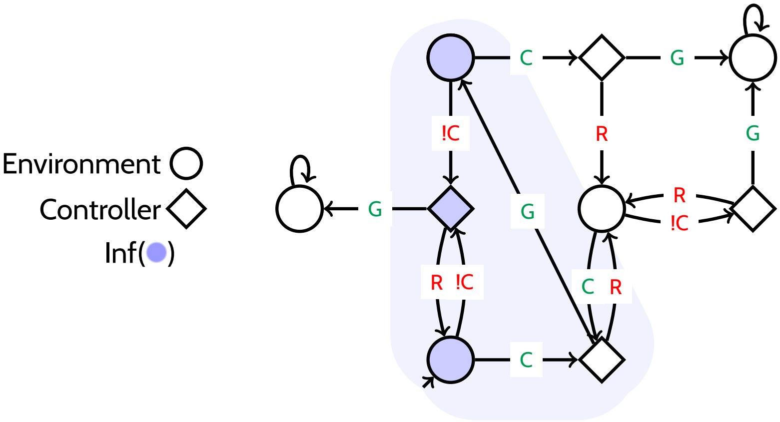


Figure: Solving a Büchi game

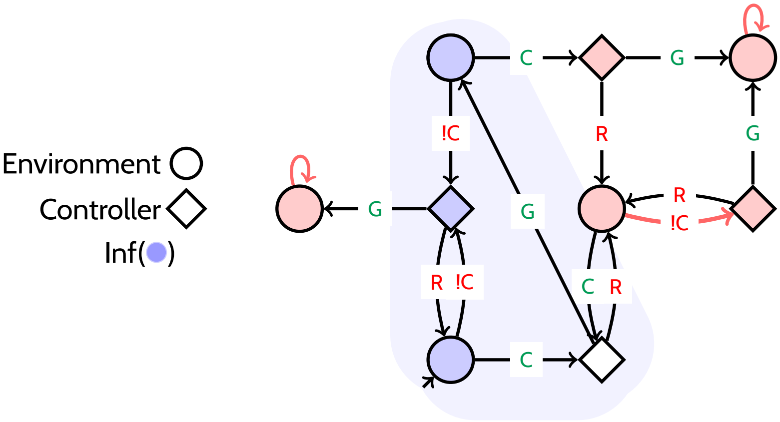


Figure: Solving a Büchi game

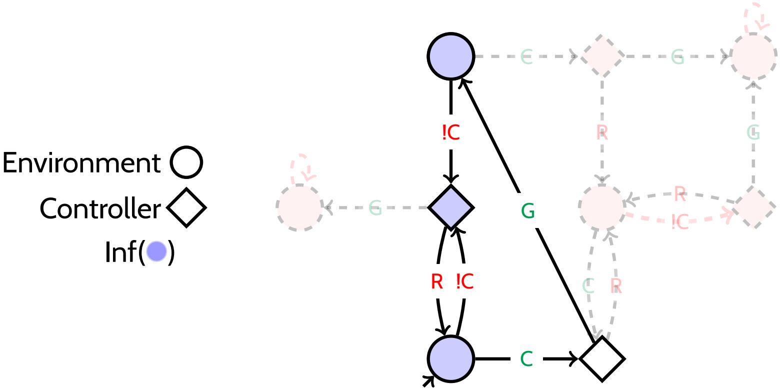


Figure: Solving a Büchi game

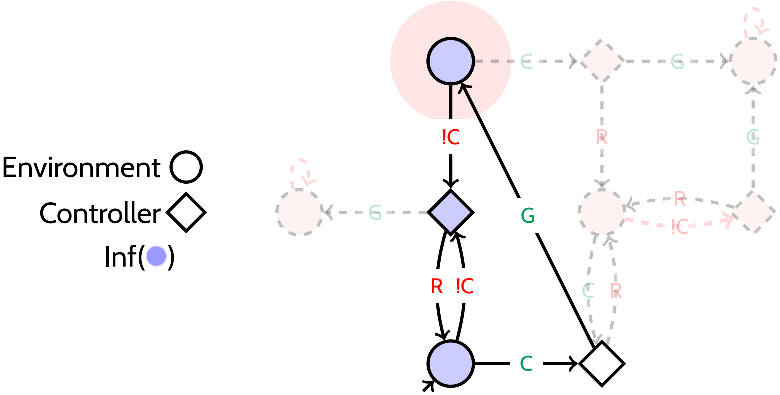


Figure: Solving a Büchi game

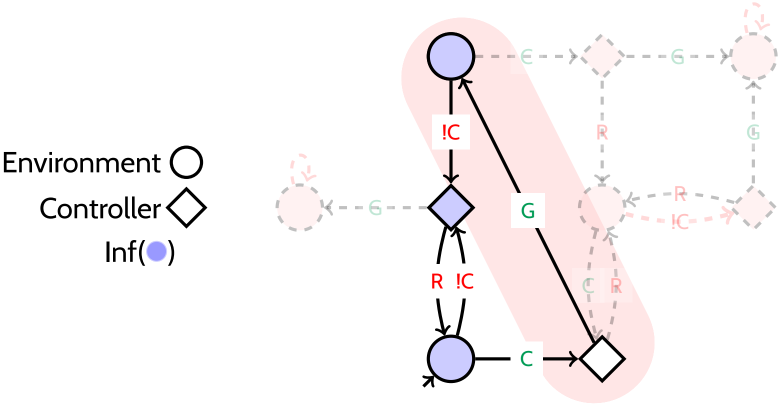


Figure: Solving a Büchi game

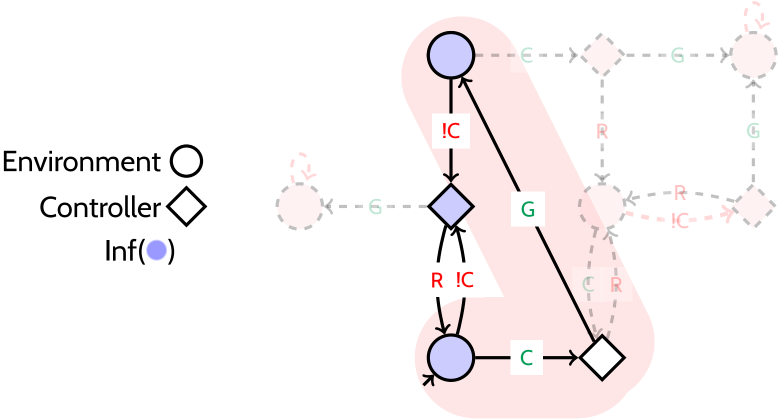


Figure: Solving a Büchi game

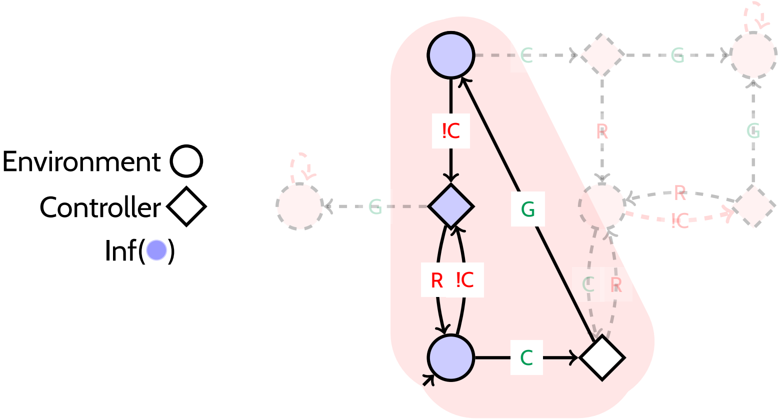


Figure: Solving a Büchi game

A smart traffic light

`ltlsynt` outline

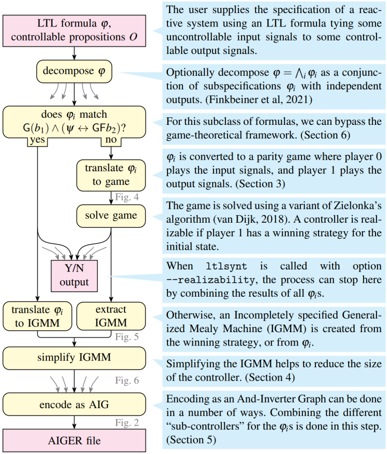


Figure: General outline of the process used by `ltlsynt` to solve the reactive synthesis problem. [2]

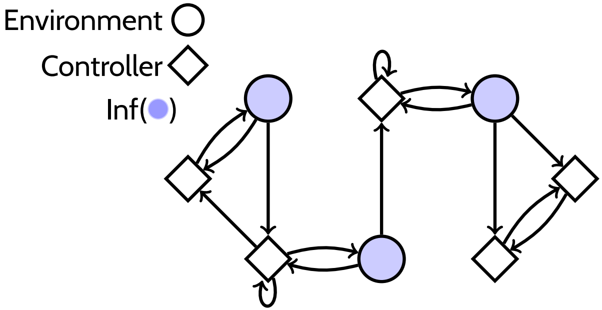


Figure: Solving a Büchi game

1 $R = \texttt{Attr}_1(B)$

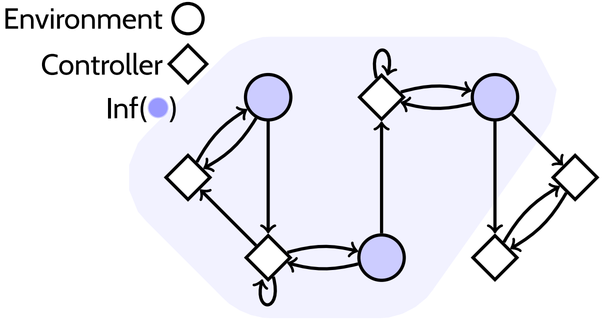


Figure: Solving a Büchi game

1. $R = \texttt{Attr}_1(B)$
2. $L = \texttt{Attr}_O(\bar{R})$

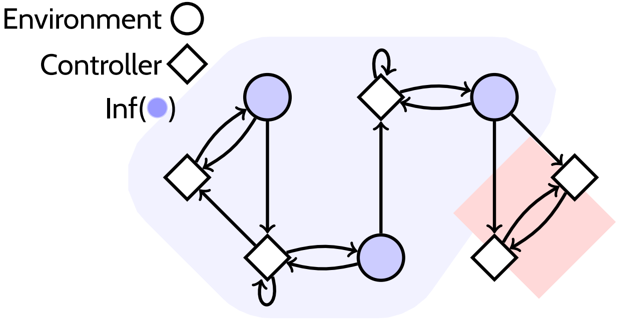


Figure: Solving a Büchi game

1. $R = \texttt{Attr}_1(B)$
2. $L = \texttt{Attr}_0(\bar{R})$

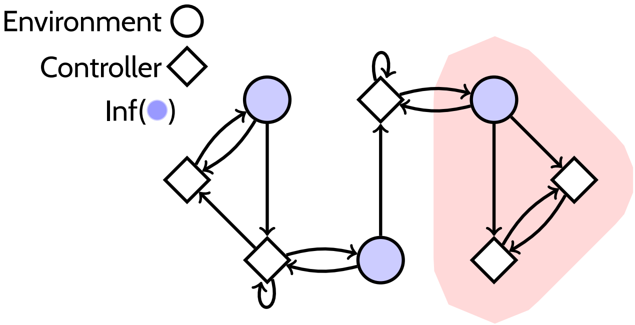


Figure: Solving a Büchi game

1. $R = \mathtt{Attr}_1(B)$
2. $L = \mathtt{Attr}_0(\bar{R})$
3. $G = G / L$

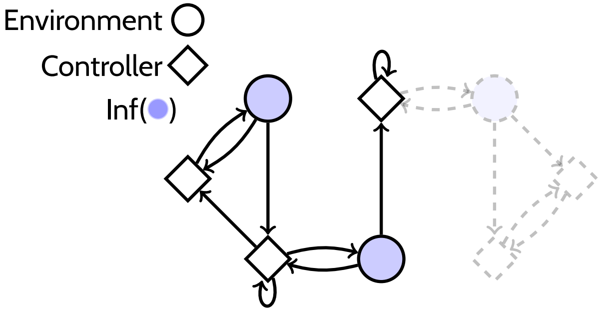


Figure: Solving a Büchi game

1. $R = \mathtt{Attr}_1(B)$
2. $L = \mathtt{Attr}_0(\bar{R})$
3. $G = G/L$
4. Repeat until fixed point is reached

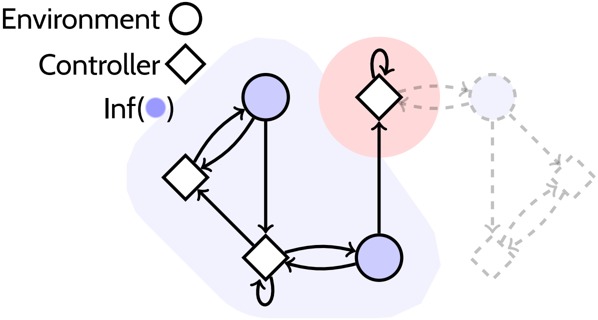


Figure: Solving a Büchi game

1. $R = \texttt{Attr}_1(B)$
2. $L = \texttt{Attr}_0(\bar{R})$
3. $G = G/L$
4. Repeat until fixed point is reached

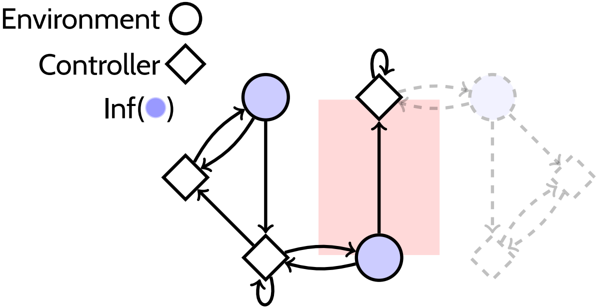


Figure: Solving a Büchi game

1. $R = \mathtt{Attr}_1(B)$
2. $L = \mathtt{Attr}_0(\bar{R})$
3. $G = G/L$
4. Repeat until fixed point is reached

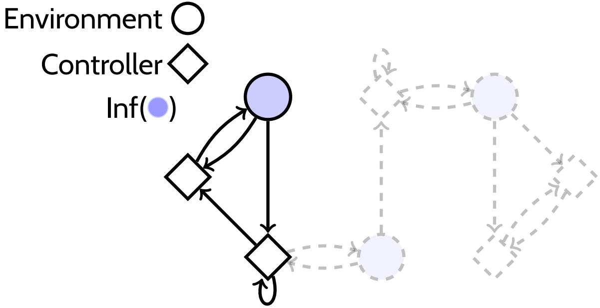


Figure: Solving a Büchi game

1. $R = \mathtt{Attr}_1(B)$
2. $L = \mathtt{Attr}_0(\bar{R})$
3. $G = G/L$
4. Repeat until fixed point is reached

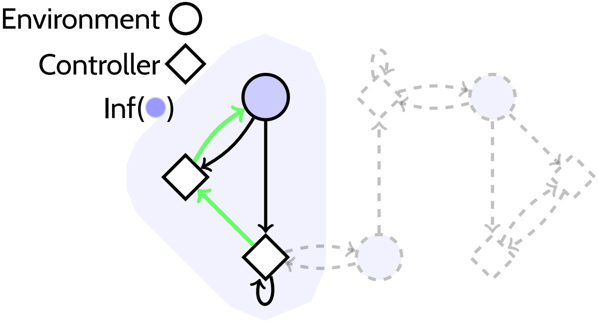


Figure: Solving a Büchi game

The data structure used to represent an arena needs to support:

- Deletion of arbitrary states and edges;
- Access to the data associated with any arbitrary state/edge;
- Traversal of all states/edges that are not yet removed.

The data structure used to represent an arena needs to support:

- Deletion of arbitrary states and edges;
- Access to the data associated with any arbitrary state/edge;
- Traversal of all states/edges that are not yet removed.

	`set`	`unordered_set`
Deletion	$O(\log(\|A\|))$	$O(1)$
Access	$O(\log(\|A\|))$	$O(1)$
Traversal	$O(\|A\|)$	$O(\|A\|)$

Where $|A|$ is the size of the *active* arena, and n is the total number of states.

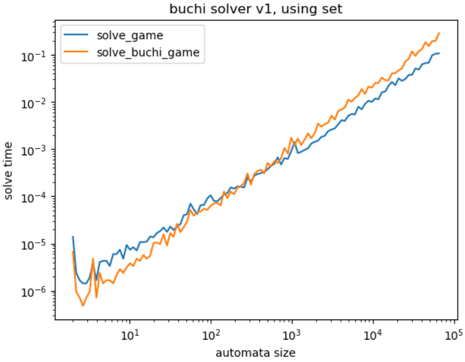


Figure: Using a `set` to represent an arena

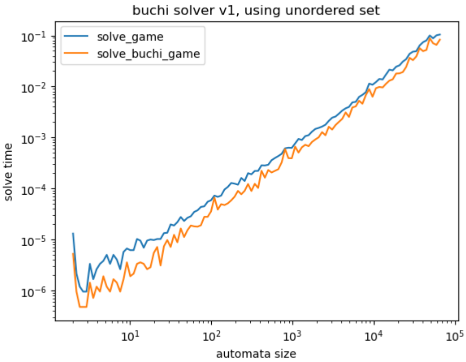


Figure: Using an `unordered_set` to represent an arena `unordered_set` have an ideal time complexity. However, they hide a strong hidden constant.

`unordered_set` have an ideal time complexity. However, they hide a strong hidden constant.

	`set`	`unordered_set`	`vector<bool>`
Deletion	$O(\log(\|A\|))$	$O(1)$	$O(1)$
Access	$O(\log(\|A\|))$	$O(1)$	$O(1)$
Traversal	$O(\|A\|)$	$O(\|A\|)$	$O(n)$

Where $|A|$ is the size of the *active* arena, and n is the total number of states.

Customized Data Structure for Büchi Solving

`unordered_set` vs. `vector<bool>`

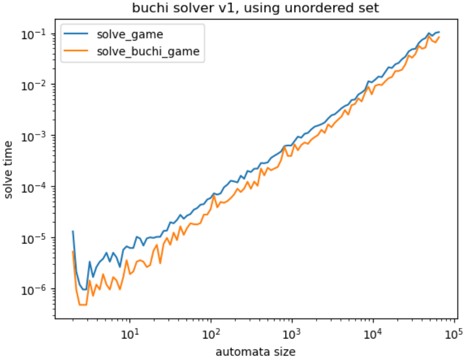


Figure: Using an `unordered_set` to represent an arena

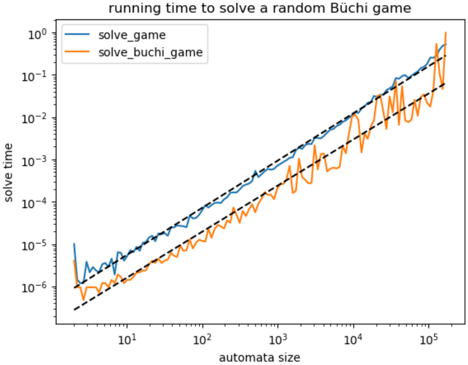
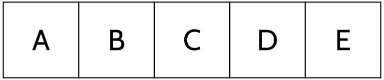


Figure: Using a `vector<bool>` to represent an arena

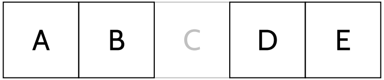
Customized Data Structure for Büchi Solving

Introducing `partitioned_dlist`

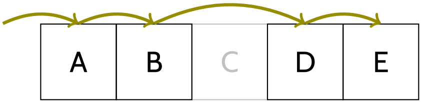
- We know in advance all the states we need to store, so we can store them statically.



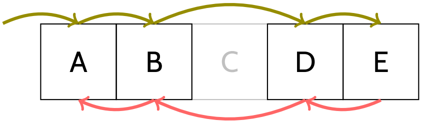
- We know in advance all the states we need to store, so we can store them statically.
- We can mark a state if it is deleted using a single `boolean`



- We know in advance all the states we need to store, so we can store them statically.
- We can mark a state if it is deleted using a single `boolean`
- To allow fast iteration over active elements, we use a pointer `next` to the next active element.



- We know in advance all the states we need to store, so we can store them statically.
- We can mark a state if it is deleted using a single `boolean`
- To allow fast iteration over active elements, we use a pointer `next` to the next active element.
- Deleting an element means the next of our previous becomes our next (we skip the element), so we also need a pointer `prev`.



Customized Data Structure for Büchi Solving

Complexity of `partitioned_dlist`

	`set`	`unordered_set`	`vector<bool>`	`partitioned_dlist`
Deletion	$O(\log(\|A\|))$	$O(1)$	$O(1)$	$O(1)$
Access	$O(\log(\|A\|))$	$O(1)$	$O(1)$	$O(1)$
Traversal	$O(\|A\|)$	$O(\|A\|)$	$O(n)$	$O(\|A\|)$

Where $|A|$ is the size of the *active* arena, and n is the total number of states.

Customized Data Structure for Büchi Solving

`vector<bool>` vs. `partitioned_dlist`

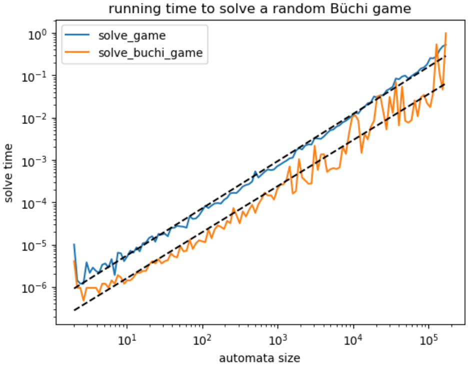


Figure: Using a `vector<bool>` to represent an arena

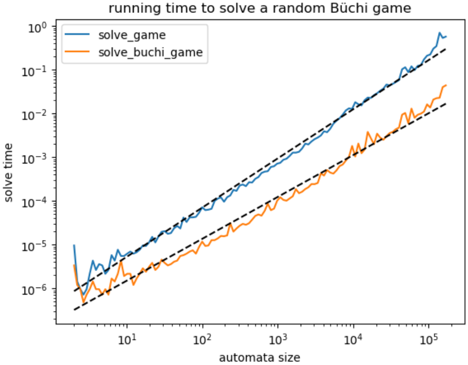


Figure: Using a `partitioned_dlist` to represent an arena

Actual state of the Büchi Solver

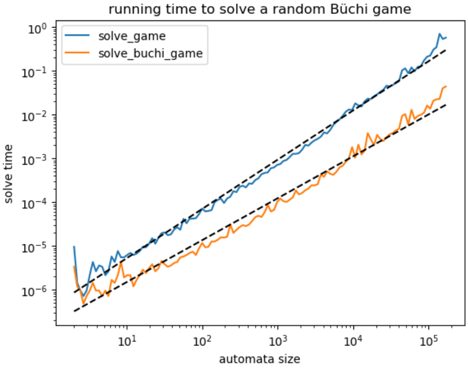


Figure: In double-log scale

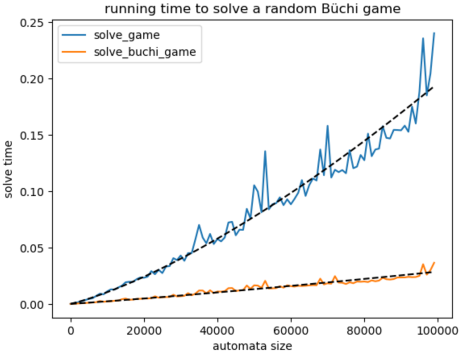


Figure: In linear scale

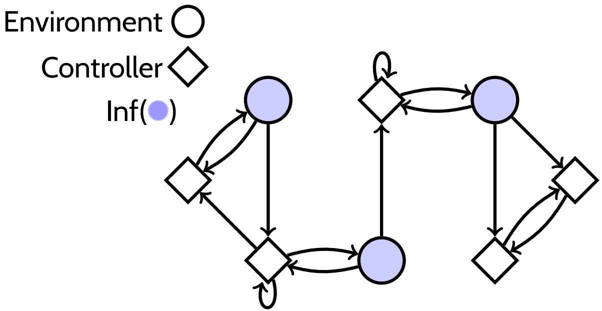


Figure: Decomposing an arena into strongly connected components

1. The solving time complexity may degenerate to $O(n^2)$

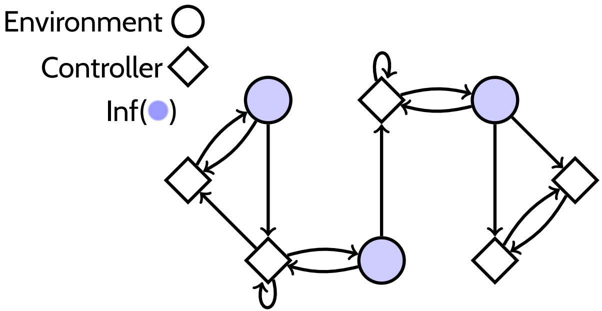


Figure: Decomposing an arena into strongly connected components

1. The solving time complexity may degenerate to $O(n^2)$
2. Decomposing into strongly connected components may help

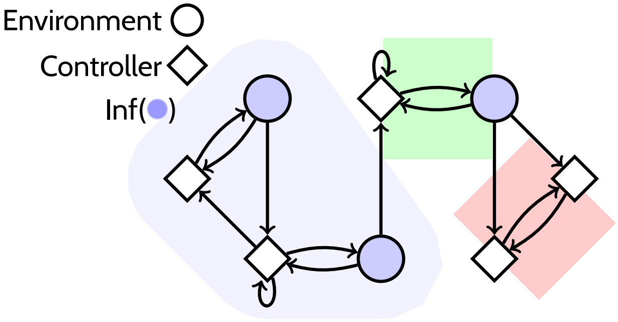
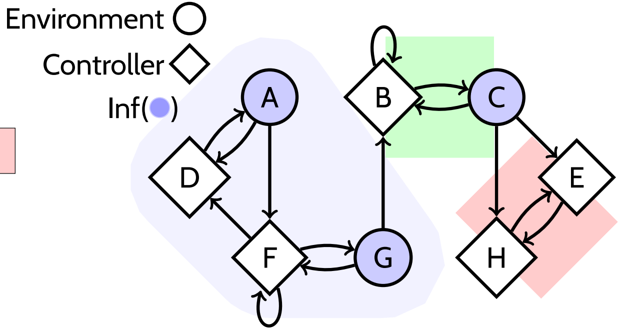


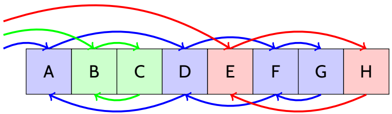
Figure: Decomposing an arena into strongly connected components

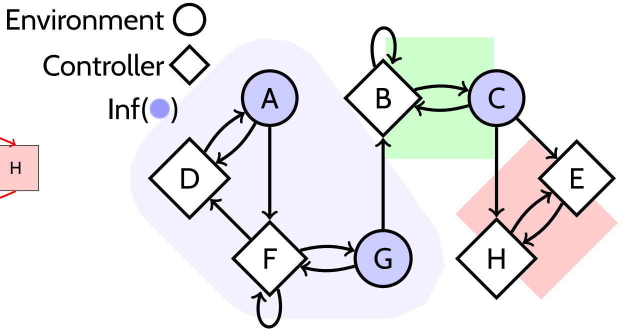
How to adapt our data structure to efficiently work only on some components?

A	B	C	D	E	F	G	H

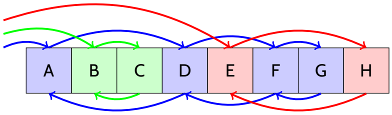


How to adapt our data structure to efficiently work only on some components?

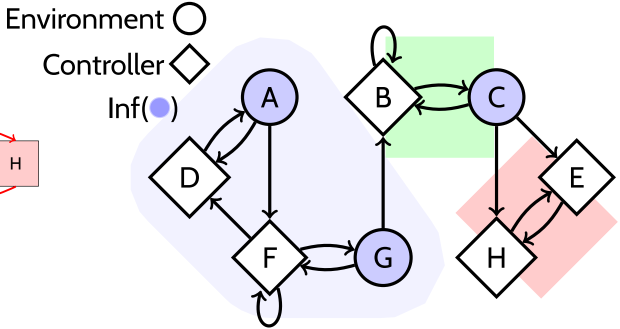




How to adapt our data structure to efficiently work only on some components?



Multiple heads!



References

Krishnendu Chatterjee, Thomas A Henzinger, and Nir Piterman.
Algorithms for büchi games.
2008.

Florian Renkin, Philipp Schlehuber-Caissier, Alexandre Duret-Lutz, and Adrien Pommellet.
Dissecting ltlsynt.
Formal Methods in System Design, 61(2):248–289, 2022.

Martin Zimmermann, Felix Klein, and Alexander Weinert.
Infinite games.
2016.