What is Model-Cheking?
How to build a Model Cheker?

Etienne Renault

2018, December 12th

https://www.lrde.epita.fr/~renault/teaching/algorep/

Foreword

Today we will build a model checker !

At the end of the day, you will be able :

Foreword

Today we will build a model checker !

At the end of the day, you will be able :

- to express properties using LTL

Foreword

Today we will build a model checker !

At the end of the day, you will be able :

- to express properties using LTL
- to understand how a (basic) model-checker works

Foreword

Today we will build a model checker !

At the end of the day, you will be able :

- to express properties using LTL
- to understand how a (basic) model-checker works
- to create a model, i.e an abstract representation of a system

Foreword

Today we will build a model checker !

At the end of the day, you will be able :

- to express properties using LTL
- to understand how a (basic) model-checker works
- to create a model, i.e an abstract representation of a system
- to check if the model meets the specification, i.e. if the system behaves as expected

Foreword

Today we will build a model checker !

At the end of the day, you will be able :

- to express properties using LTL (see previous lesson)
- to understand how a (basic) model-checker works
- to create a model, i.e an abstract representation of a system
- to check if the model meets the specification, i.e. if the system behaves as expected

What is a system ?

All theses pictures are under CreativeCommons

Why a model is required ?

The following server-like snippet can be considered as a system.

```
unsigned received_ = 0;
while (1)
{
  accept_request();
  received_ = received_ + 1;
  reply_request();
}
```

How many configurations for such a program ?

We have 2 unsigned variables (received_ + Program Counter).
In the worst case : $(2^{32} - 1)^2$

What is a model ?

Real systems have hundreds of thousands variables !

Since model checker may explore all these configurations, we must reduce the memory complexity.

A model is an abstract representation of the system

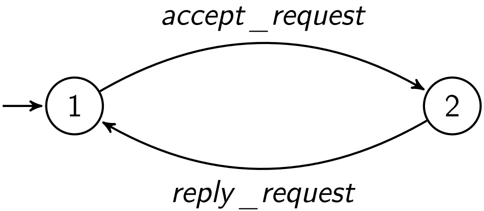
- A model has less variables than the real system
- A model has less *configurations* than the real system
- A model mostly focuses on behaviors and interactions
- A model has **a finite number of variables**, i.e. no dynamic allocations

How to represent a model?

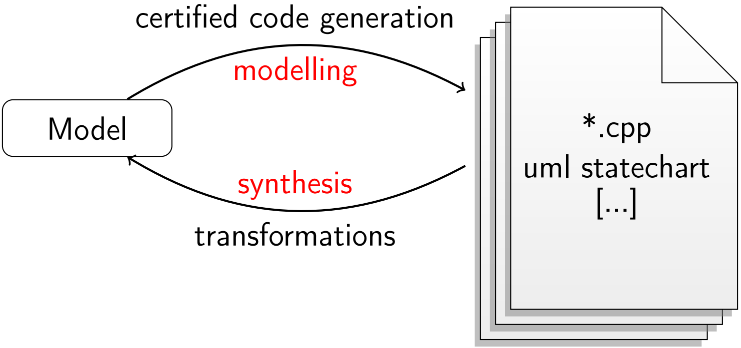
Each component of the system can be represented like an finite state automaton

- possible only since there is a finite number of finite size variables

The previous server-like snippet can then be represented as following :



How to build a model ?



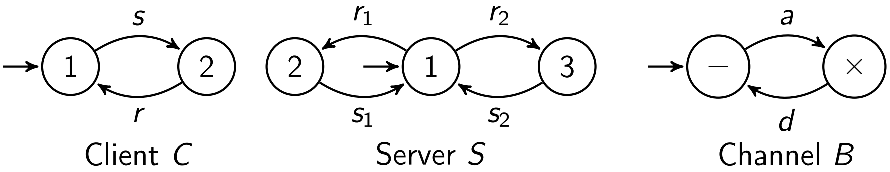
Model formalisms

There are a lot of formalisms :

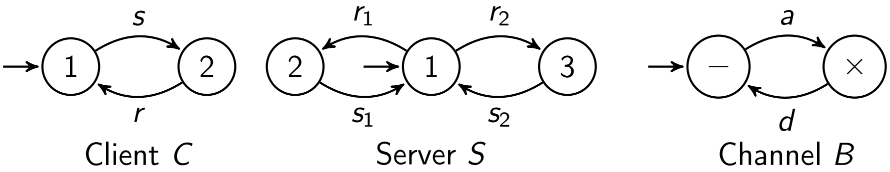
- PetriNet, Fiacre, **DVE**, Promela, AADL, etc.

All are not equivalent but there are all formally specified.

A more realistic example !

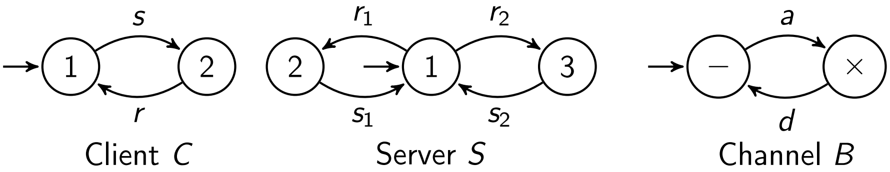


A more realistic example !



1 server, 2 clients, 4 channels

A more realistic example !

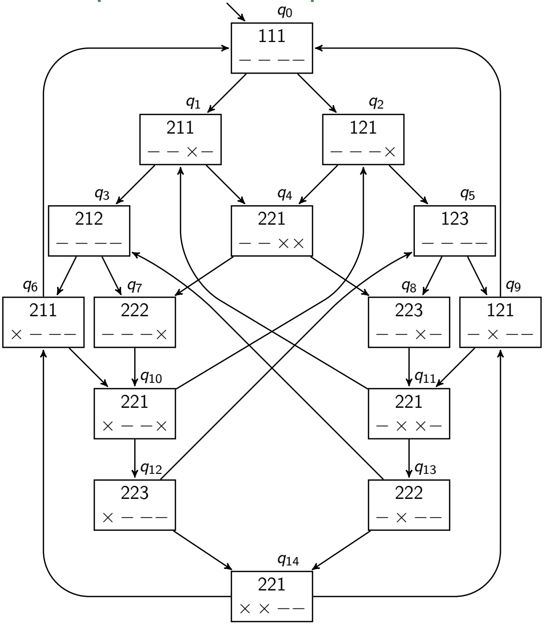


1 server, 2 clients, 4 channels

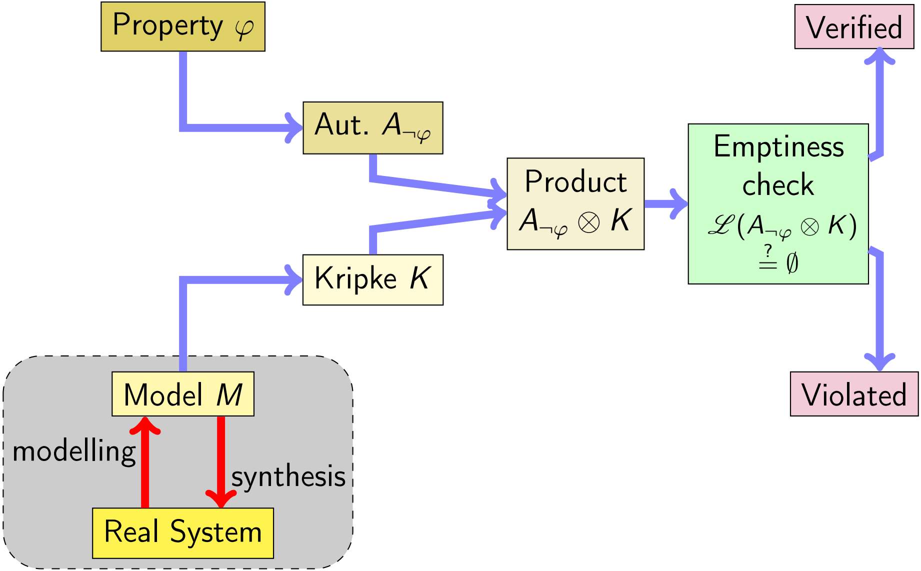
System's synchronization rules $\langle C, C, S, B, B, B, B \rangle$:

(1)	s	.	.	.	.	a	.
(2)	.	s	.	.	.	.	a
(3)	r	.	.	d	.	.	.
(4)	.	r	.	.	d	.	.
(5)	.	.	r_1	.	.	d	.
(6)	.	.	s_1	a	.	.	.
(7)	.	.	r_2	.	.	.	d
(8)	.	.	s_2	.	a	.	.

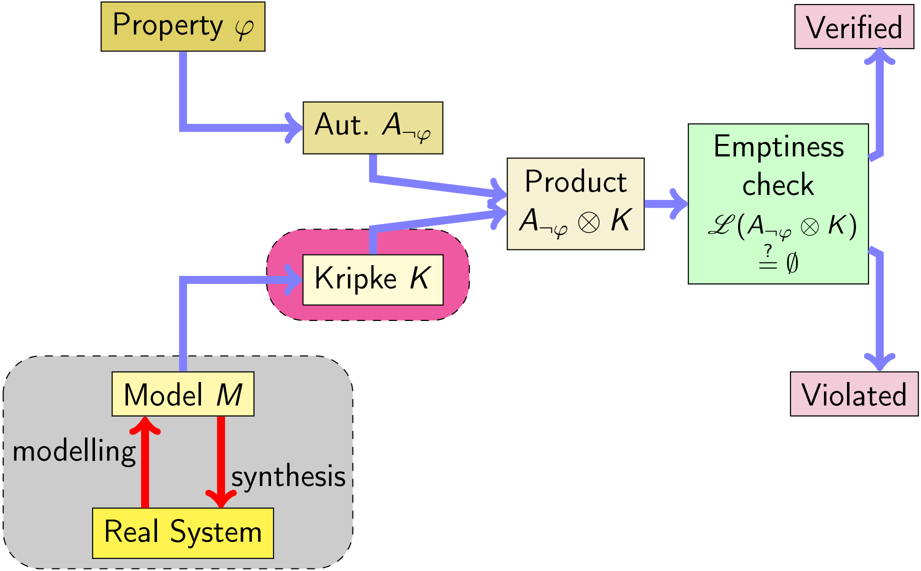
Example's state space



Automata approach for model checking



Automata approach for model checking



Kripke structure

State machine labelled by atomic propositions.

A Kripke structure is a 5 tuple $K = \langle AP, \mathcal{Q}, q^0, \delta, l \rangle$ with

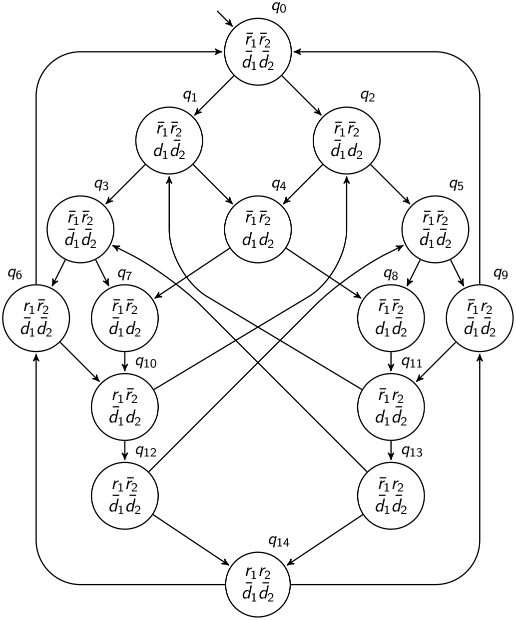
- AP is the set of atomic propositions
- $\mathcal{Q}$ is the finite set of state
- $q^0 \in \mathcal{Q}$ is the initial state
- $\delta : \mathcal{Q} \mapsto 2^{\mathcal{Q}}$ is the transition function that associates successors to a given state
- $l : \mathcal{Q} \mapsto 2^{AP}$ is labelling function that associates atomic propositions to a given state

Atomic propositions for the example

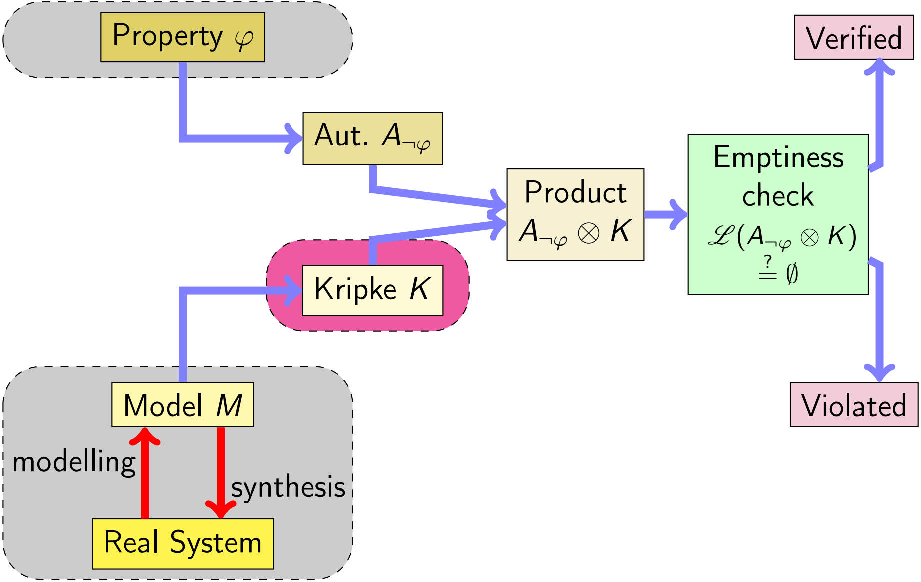
We want to track messages received and sent. Let us define $AP = \{r_1, r_2, d_1, d_2\}$, s.t. :

- r_1 : a response is in progress between the server and the first client
- r_2 : a response is in progress between the server and the second client
- d_1 : a request (d for demand) is in progress between the first client and the server
- d_2 : a request (d) is in progress between the second client and the server

Kripke Structure for the example



Automata approach for model checking



How to express Infinite behavior ?

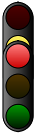
Propositionnal Logic : the present instant

r : Red traffic light on

o : Orange traffic light on

v : Green traffic light on

$r \wedge o \wedge v =$ , $r \wedge \neg o \wedge \neg v =$ , $\neg r \wedge \neg o \wedge v =$ , $\neg r \wedge \neg o \wedge \neg v =$ .

How to say that  happens before  ?
How to say that stay forever ?

$\Rightarrow$ Time must be expressed

LTL : Linear Temporal Logic

BNF

$$\varphi ::= \top \mid \bot \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \,\mathbf{U}\, \psi \mid \mathbf{X}\, \varphi$$

Syntaxic Sugar

$$\mathbf{F}\,\varphi \equiv \top\,\mathbf{U}\,\varphi$$
$$\varphi\,\mathbf{R}\,\psi \equiv \neg(\neg\varphi\,\mathbf{U}\,\psi)$$
$$\mathbf{G}\,\varphi \equiv \bot\,\mathbf{R}\,\varphi$$
$$\varphi\,\mathbf{W}\,\psi \equiv \psi\,\mathbf{R}(\varphi \vee \psi)$$

Globally

Meaning : $w \models \mathbf{G}\,\varphi \Longleftrightarrow \forall i, w_i \models \varphi$

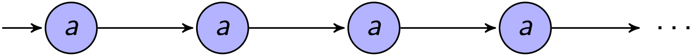
Explanations : Propety f is satisfied all along w iff any subwords of w satisfies φ

Globally

Meaning : $w \models \mathbf{G}\,\varphi \iff \forall i, w_i \models \varphi$

Explanations : Propety f is satisfied all along w iff any subwords of w satisfies φ

Système vérifiant : $\mathbf{G}\,a$



Finally

Meaning : $w \models F\varphi \Longleftrightarrow \exists i, w_i \models \varphi$

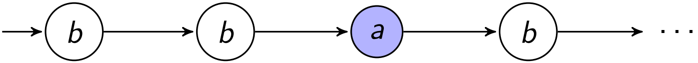
Explanation : f is satisfied at least once along the path c iff one of the sub-path of c satisfies f

Finally

Meaning : $w \models F\varphi \Longleftrightarrow \exists i, w_i \models \varphi$

Explanation : f is satisfied at least once along the path c iff one of the sub-path of c satisfies f

System satisfying : **F** a



Next

Meaning : $w \models X\varphi \iff c_1 \models \varphi$

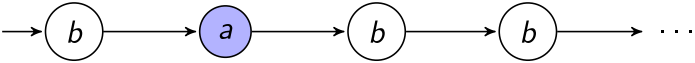
Explanation : Property φ is satisfied par the successors of state w

Next

Meaning : $w \models X\varphi \iff c_1 \models \varphi$

Explanation : Property φ is satisfied par the successors of state w

System satisfying : $\mathbf{X}\ a$



Until

Meaning : $c \models fUg \iff \exists i, c_i \models g \wedge \forall j < i, c_j \models f$

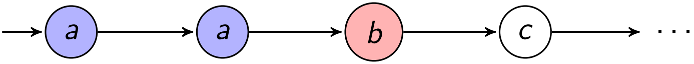
Explanation : from a given step of the path c all sub-paths satisfy g, and f is satified from all preceding sub-pathes

Until

Meaning : $c \models fUg \iff \exists i, c_i \models g \wedge \forall j < i, c_j \models f$

Explanation : from a given step of the path c all sub-paths satisfy g, and f is satified from all preceding sub-pathes

System satisfying : $a\,\mathbf{U}\,b$



Weak until

Sémantique : $w \models fWg \iff fUg \vee Gf$

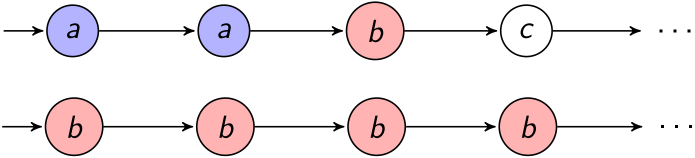
Explication : si g est vérifiée à partir d'une certaine étape du chemin, alors f a été vérifiée tout au long du chemin précédant

Weak until

Sémantique : $w \models fWg \iff fUg \lor Gf$

Explication : si g est vérifiée à partir d'une certaine étape du chemin, alors f a été vérifiée tout au long du chemin précédant

Système vérifiant : $a\,\mathbf{W}\,b$



LTL : Linear Time temporal Logic

Equivalent to F1S

$\neg\,\mathbf{G}(r \wedge \neg o \wedge \neg v)$: the system is not always .

$\mathbf{G}((\neg r \wedge o \wedge \neg v) \rightarrow \mathbf{X}(r \wedge \neg o \wedge \neg v))$:  is always followed by .

$\mathbf{G}\,\mathbf{F}(\neg r \wedge \neg o \wedge v)$: The system is infinitly often .

LTL : Linear Time temporal Logic

F, **G** et **R** (Release) are syntaxic sugar :

$$\mathbf{F}\, f = \top\, \mathbf{U}\, f$$

$$f\, \mathbf{R}\, g = \neg(\neg f\, \mathbf{U}\, \neg g)$$

$$\mathbf{G}\, f = \neg\, \mathbf{F}\, \neg f = \neg(\top\, \mathbf{U}\, \neg f) = \bot\, \mathbf{R}\, f$$

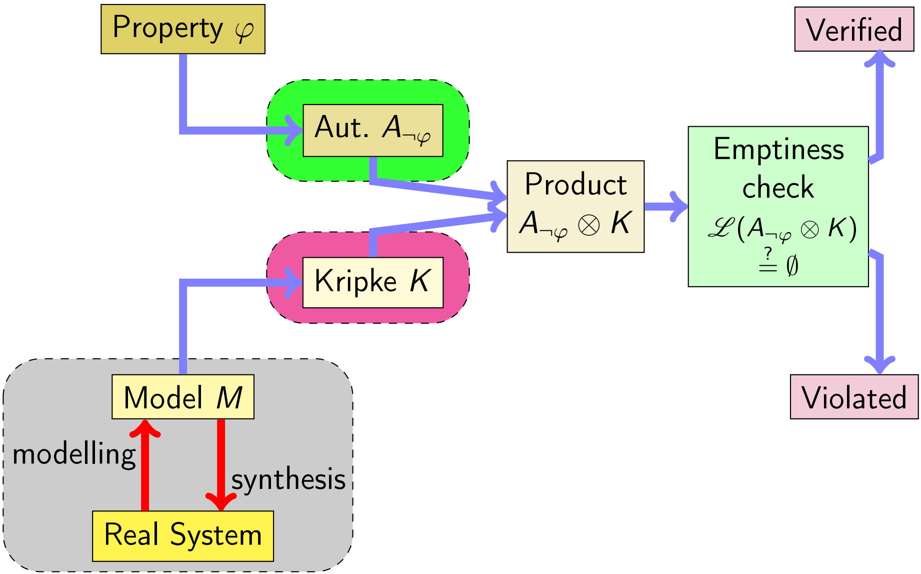
We have also :

$$\neg\, \mathbf{X}\, f = \mathbf{X}\, \neg f$$

$$\neg\, \mathbf{F}\, f = \mathbf{G}\, \neg f \qquad \neg(f\, \mathbf{U}\, g) = (\neg f)\, \mathbf{R}(\neg g)$$

$$\neg\, \mathbf{G}\, f = \mathbf{F}\, \neg f \qquad \neg(f\, \mathbf{R}\, g) = (\neg f)\, \mathbf{U}(\neg g)$$

Automata approach for model checking

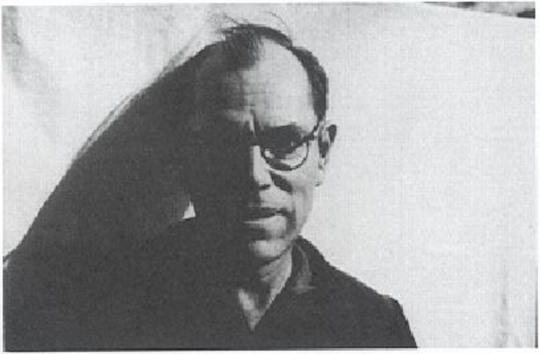


Converting LTL into Something Else

LTL is a text representation : this is not very friendly to manipulate

How to represent something that accepts a word (a sequence or a run of the system)

Julius Richard Büchi (1924–1984)



J. Richard Büchi, 1983

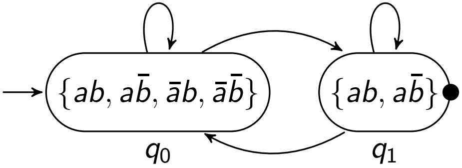
Logicien et mathématicien suisse.
Phd in Zürich [1950], move then to the USA.
Showed decidability of S1S.

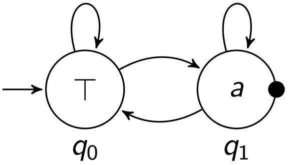
Automates de Büchi

A Büchi automaton is a 6-uplet $A = \langle \Sigma, \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \delta, l \rangle$ où

- Σ the alphabet,
- $\mathcal{Q}$ a finite set of states,
- $\mathcal{Q}^0 \subseteq \mathcal{Q}$ a subset of initial states,
- $\mathcal{F} \subseteq \mathcal{Q}$ a set of accepting states,
- $\delta : \mathcal{Q} \mapsto 2^{\mathcal{Q}}$ the transition relation,
- $l : \mathcal{Q} \mapsto 2^{\Sigma} \setminus \{\emptyset\}$ labels each state with a (non empty) set of letters

Example with $AP = \{a, b\}$, $\Sigma = 2^{AP}$:





Büchi Automata : langages

Les chemins de A :

$$\text{Run}(A) = \{q_0 \cdot q_1 \cdot q_2 \cdots \in \mathcal{Q}^\omega \mid q_0 \in \mathcal{Q}^0 \text{ et } \forall i \geq 0,\ q_{i+1} \in \delta(q_i)\}$$

Accepting runs of A are thoses that visit infinitely often accepting states :

$$\text{Acc}(A) = \{r \in \text{Run}(A) \mid \forall i \geq 0, \exists j \geq i,\ r(j) \in \mathcal{F}\}$$

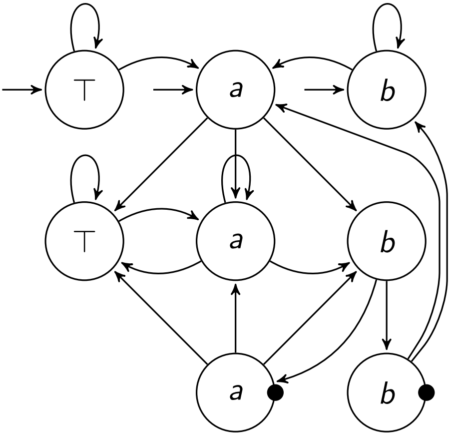
A run of A is a sequence $\sigma \in \Sigma^\omega$ for which there is an accepting path $q_0 \cdot q_1 \cdots \in \text{Acc}(A)$ where labels contain letters of : $\forall i \in \mathbb{N},\ \sigma(i) \in l(q_i)$.

The language of A is the set of executions of A :

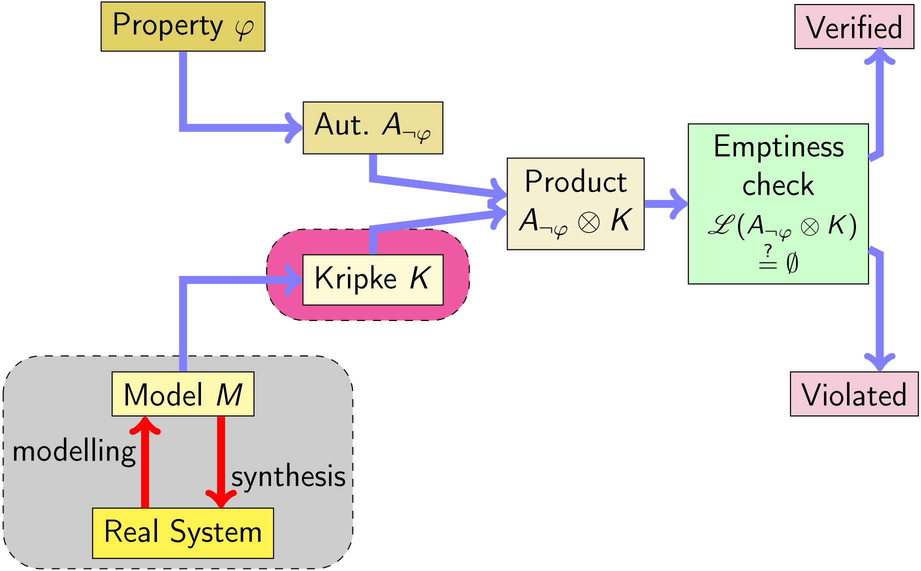
$$\mathscr{L}(A) = \{\sigma \in \Sigma^\omega \mid \exists q_0 \cdot q_1 \cdot q_2 \cdots \in \text{Acc}(A),\ \forall i \in \mathbb{N},\ \sigma(i) \in l(q_i)\}$$

More accepting states

Example with $AP = \{a, b\}$, $\Sigma = 2^{AP}$:

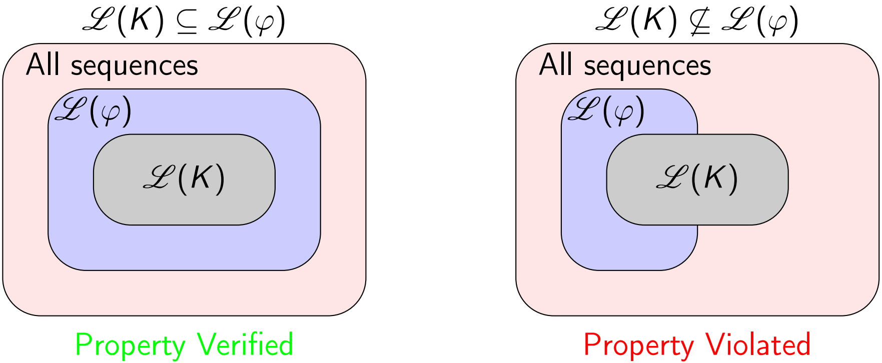


Automata approach for model checking

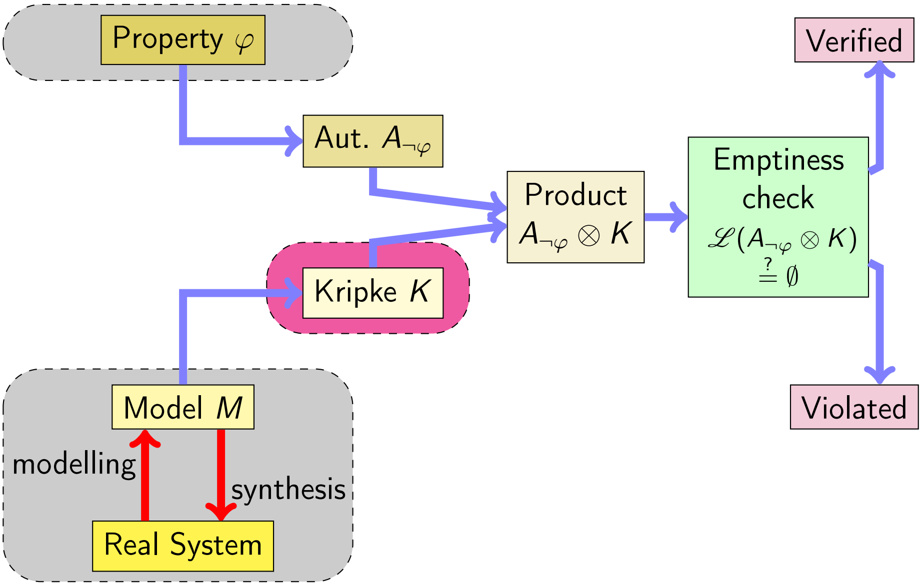


Why to check $\mathscr{L}(A_{\neg\varphi} \otimes K) \stackrel{?}{=} \emptyset$?

We want to check $\mathscr{L}(K) \subseteq \mathscr{L}(\varphi)$, which is equivalent to check $\mathscr{L}(K) \cap \overline{\mathscr{L}(\varphi)} \stackrel{?}{=} \emptyset$, which is equivalent to check $\mathscr{L}(A_{\neg\varphi} \otimes K) \stackrel{?}{=} \emptyset$



Automata approach for model checking



Express Property Automaton

How to express?

If client 1 send a request, he will necessarily receive a response

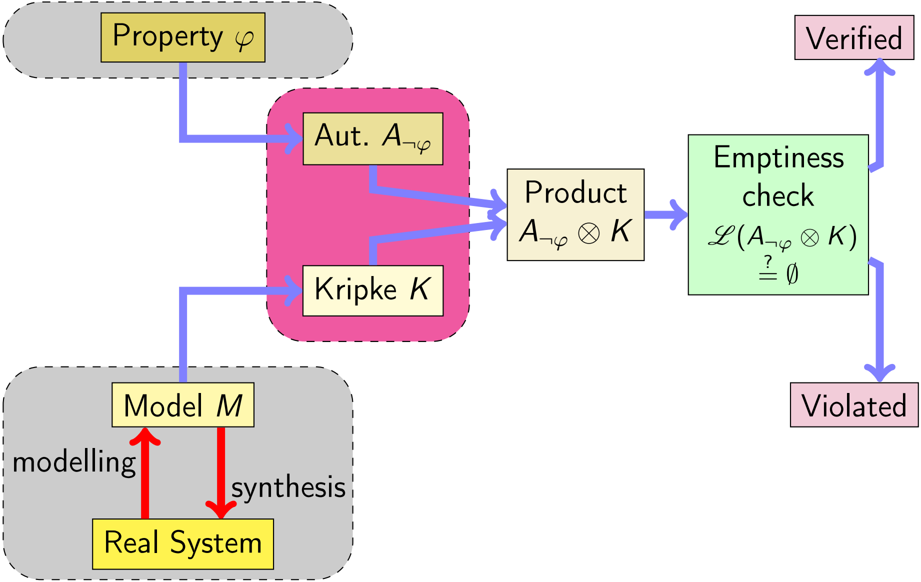
Express Property Automaton

How to express ?

If client 1 send a request, he will necessarily receive a response

'(G(d_1 -> F r_1))'

Automata approach for model checking



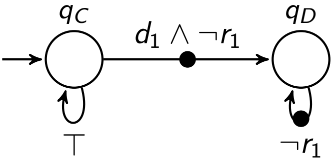
Express Property Automaton

How to express ?

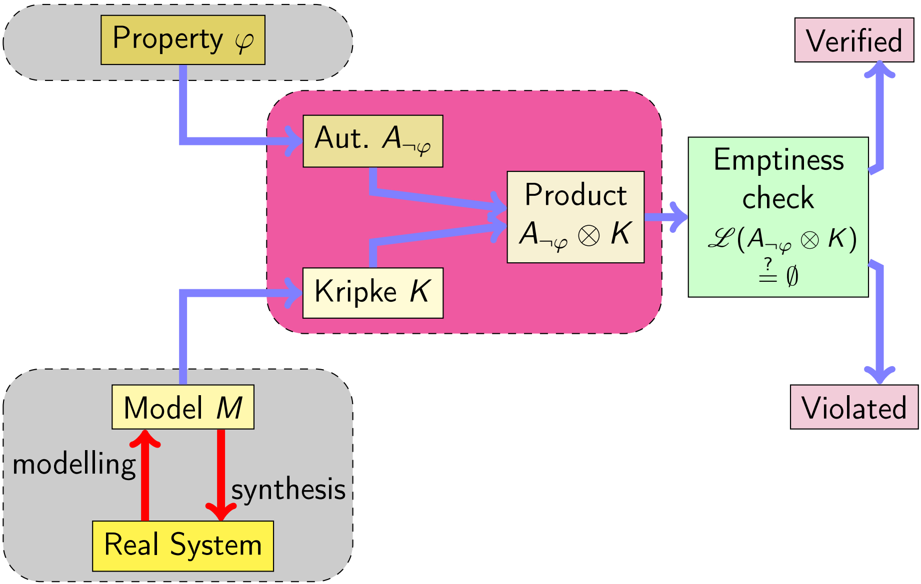
If client 1 send a request, he will necessarily receive a response

'(G(d_1 -> F r_1))'

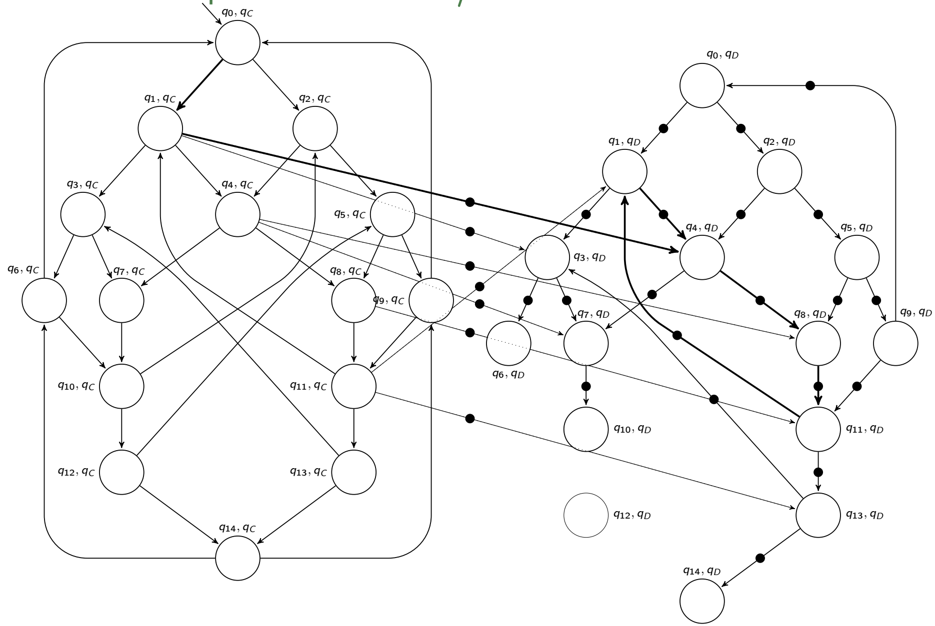
We can translate '!(G(d_1 -> F r_1))' into an automaton :



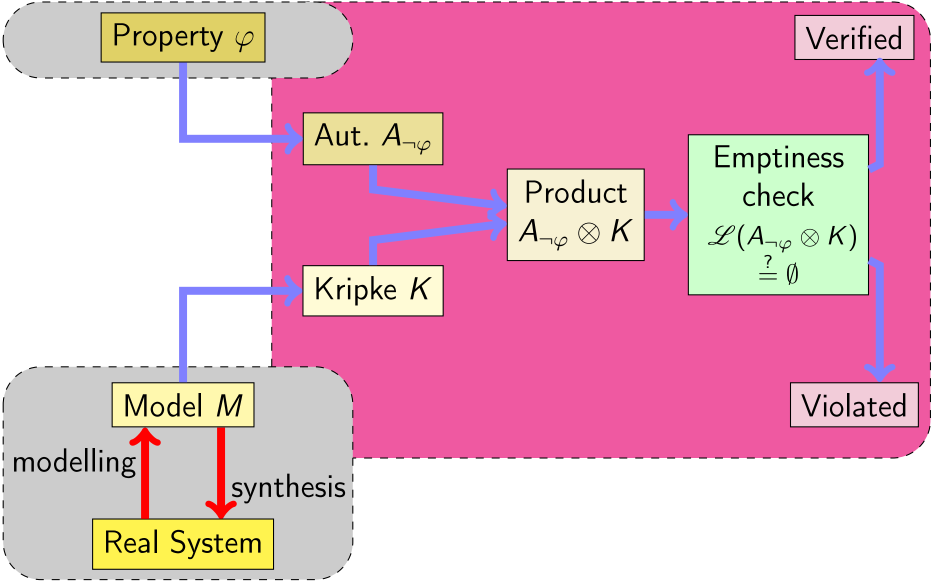
Automata approach for model checking



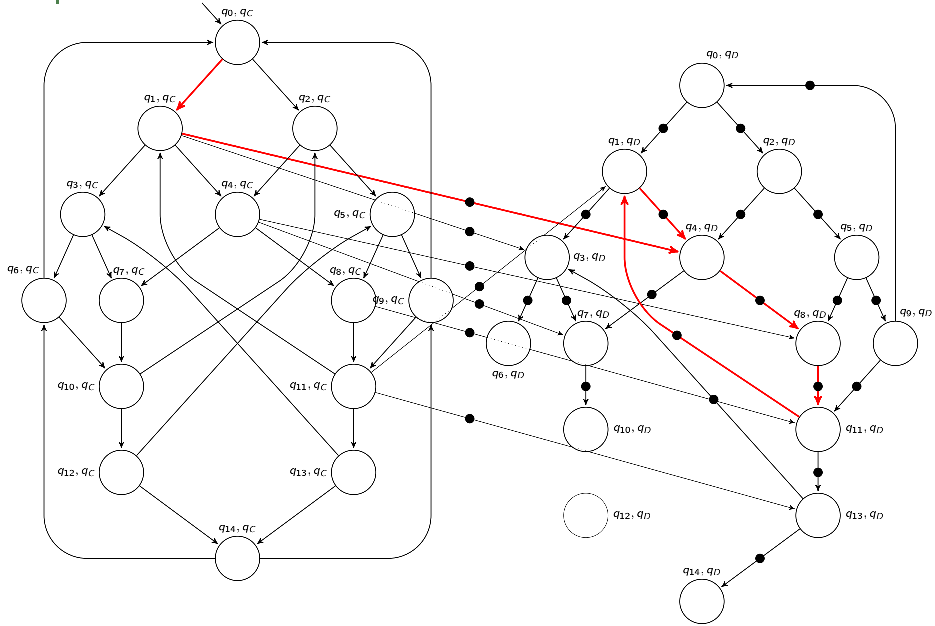
Product Kripke structure / Automaton



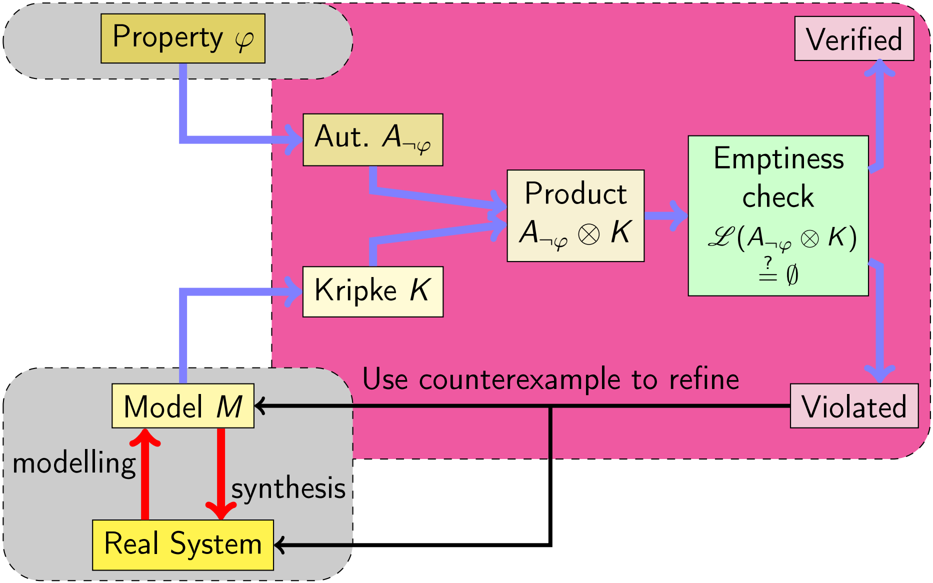
Automata approach for model checking



Emptiness check



Automata approach for model checking



Sum up

- From a model, we can build the kripke structure if :
  - we can extract the initial state
  - we can compute the successors of a given state
- Divine2.4 tool (patch by LTSmin) build such a Kripke structure
  - from the DVE language
  - spot can read kripke structures generated by Divine2.4
  - BNF for DVE can be found (page 8 – 9) at `https://is.muni.cz/www/208047/meandve.pdf`