Automates, algèbre, applications - AAA

CM 2 - model-checking

Uli Fahrenberg Sven Dziadek Philipp Schlehuber
Adrien Pommellet Etienne Renault

EPITA

S6 2022

Foreword

A simple example

```
func fibo(n int) int {
  n0 := 0
  n1 := 1
  for i := 0; i < n; i++{
    n2 := n0 + n1
    n0 = n1
    n1 = n2
  }
  return n1
}
func main() {
   a := 1
   for ; a < 10; {
       a = fibo(5)
   }
}
```

A simple example

```
func fibo(n int) int {
  n0 := 0
  n1 := 1
  for i := 0; i < n; i++{
    n2 := n0 + n1
    n0 = n1
    n1 = n2
  }
  return n1
}
func main() {
   a := 1
   for ; a < 10; {
        a = fibo(5)
   }
}
```

Question

How to ensure that this program loops infinitely?

A simple example

```
func fibo(n int) int {
  n0 := 0
  n1 := 1
  for i := 0; i < n; i++{
    n2 := n0 + n1
    n0 = n1
    n1 = n2
  }
  return n1
}
func main() {
   a := 1
   for ; a < 10; {
         a = fibo(5)
   }
}
```

Question

How to ensure that this program loops infinitely?

Answer

Use model-checking!
(very different from testing)

First (and vague) definition

What is model-cheking?

A way to **prove** (formally, mathematically) that your (infinite) *programs* are **correct**.

First (and vague) definition

What is model-cheking?

A way to **prove** (formally, mathematically) that your (infinite) *programs* are **correct**.

In other words

How to ensure that a system behaves as expected?

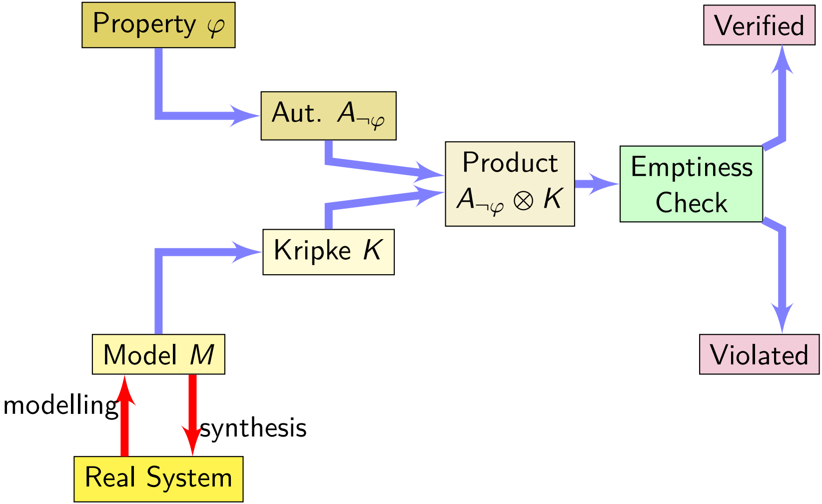
$\Rightarrow$ *Tests are bad since they only test few corner cases...*

Automata approach for model checking

Various approaches to model-checking

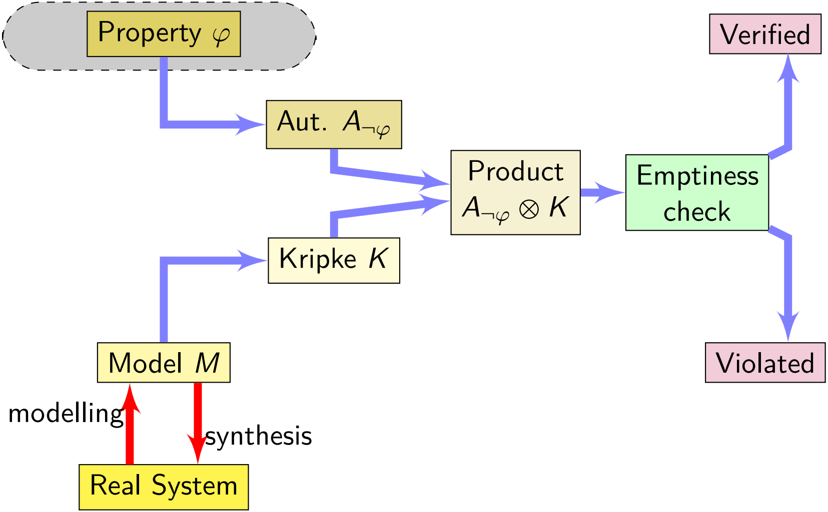
- Explicit approaches (automata-based)
- Symbolic approaches (BDD, SAT, ...)

Automata approach for model checking



Property

Automata approach for model checking



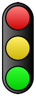
How to express Present Instant?

Propositionnal Logic: the present instant

r : Red traffic light on

o : Orange traffic light on

v : Green traffic light on

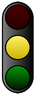
 $= r \wedge o \wedge v$

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

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

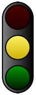
 $= \neg r \wedge \neg o \wedge \neg v$

How to express Infinite behavior?

How to say that  happens before  ?

How to say that  stay forever?

How to express Infinite behavior?

How to say that  happens before  ?

How to say that  stay forever?

$\Rightarrow$ Time must be expressed

$\Rightarrow$ This is LTL

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

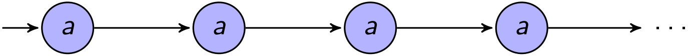
$$\begin{aligned}
\mathbf{F}\, \varphi &\equiv \top \,\mathbf{U}\, \varphi \\
\varphi \,\mathbf{R}\, \psi &\equiv \neg(\neg\varphi \,\mathbf{U}\, \psi) \\
\mathbf{G}\, \varphi &\equiv \bot \,\mathbf{R}\, \varphi \\
\varphi \,\mathbf{W}\, \psi &\equiv \psi \,\mathbf{R} (\varphi \vee \psi)
\end{aligned}$$

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

System satisfies : $\mathbf{G}\,a$



Finally

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

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

System satisfies : $\mathbf{F}\ a$

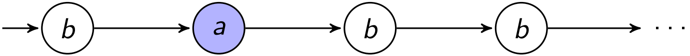
$$\rightarrow (b) \rightarrow (b) \rightarrow (a) \rightarrow (b) \rightarrow \cdots$$

Next

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

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

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

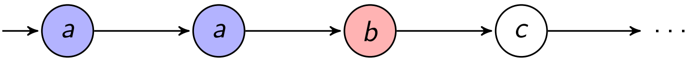


Until

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

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



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:

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

Other Logics

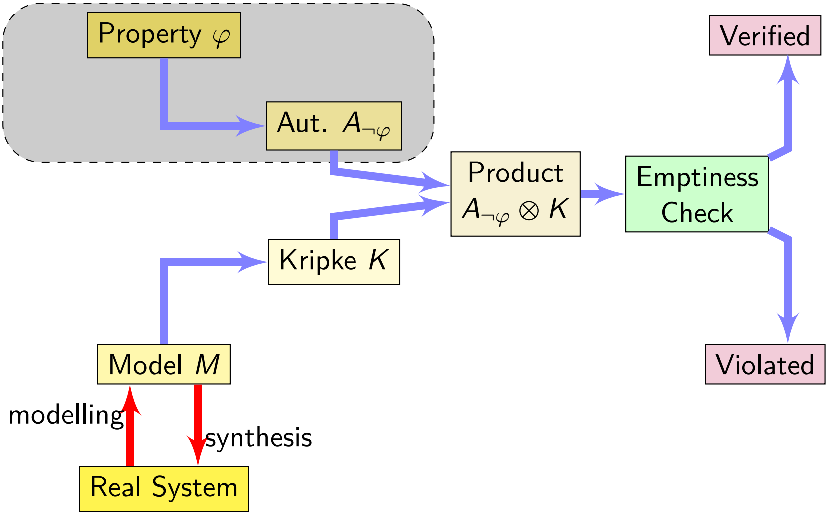
LTL is Equivalent to Monadic First-Order Logic

Other temporal logic exist

- Computation tree logic (CTL)
- CTL* which generalizes LTL and CTL
- mu-calculus
- Property specification language (PSL)
- HyperLTL
- .. and many more!

Converting LTL into something else

Automata approach for model checking

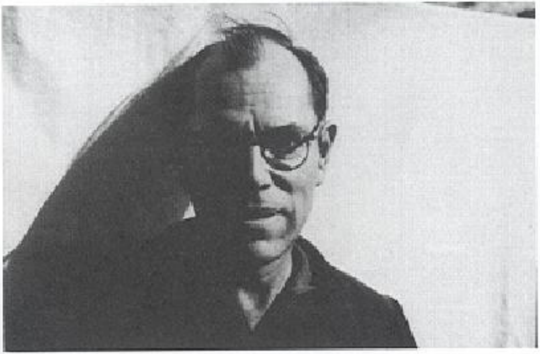


Converting LTL into Something Else

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.

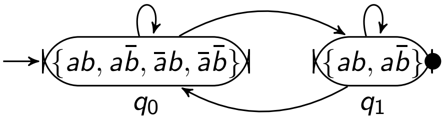
Showed decidability of S1S.

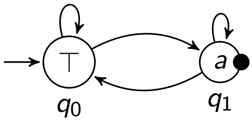
Büchi Automata

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 Runs of A:

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

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

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

A run of A is a sequence $\sigma \in \Sigma^\omega$ for which there is an accepting path $q_0 \cdot q_1 \cdots \in \mathrm{Acc}(A)$ 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:

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

Various kind of Büchi automata

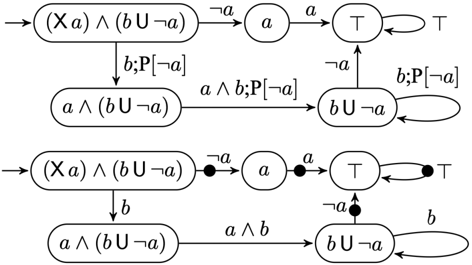
- Büchi Automata (BA)
- Transition-based Büchi Automata (TBA)
- Generalized Büchi Automata (GBA)
- Transition-based Generalized Büchi Automata (TGBA)
- Terminal, Weak, Strong Büchi Automata
- Alternating Automata
- ...

Translating LTL into an automaton

Couvreur 1999

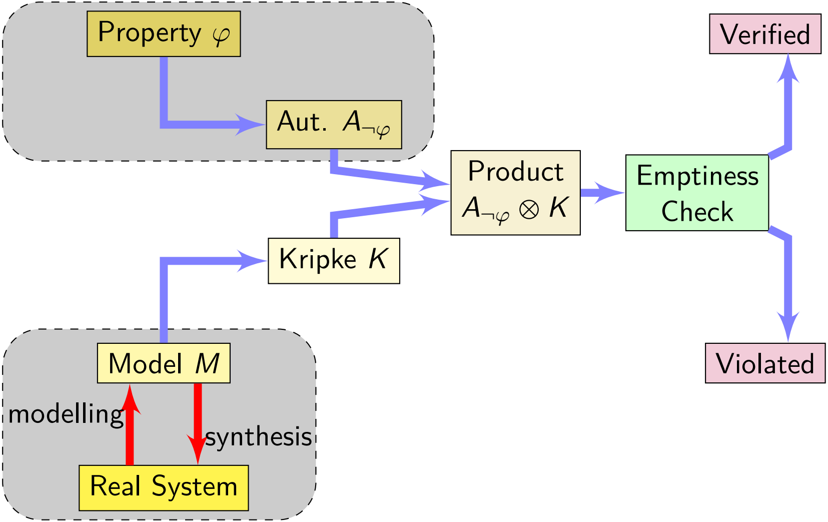
Progress in the formula until subformula cannot be postponed

- Formula rewritting
- Promise to fulfill some subformula
- indicate what is the next atom to see



Systems and Models

Automata approach for model checking



What is a system?






Why a model is required?

The following server-like C 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

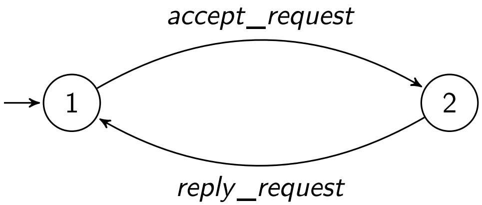
- A model has less variables than the real system
- A model has less *configurations* than the real system
- A model mostly focuses on behaviors and interactions
- A model has **a finite number of variables**, i.e. no dynamic allocations

How to represent a model?

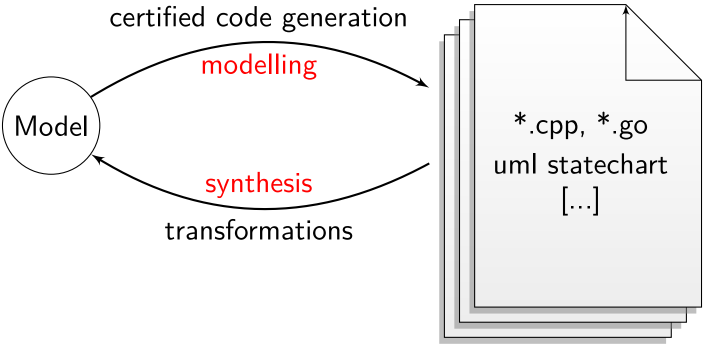
Each component of the system can be represented like an finite state automaton

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

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



How to build a model?



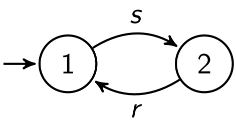
Model formalisms

There are a lot of formalisms:

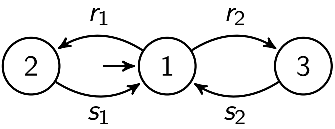
- PetriNet, Fiacre, **DVE**, Promela, AADL, etc.

All are not equivalent but there are all formally specified.

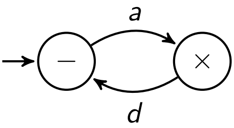
A more realistic example!



Client C



Server S



Channel B

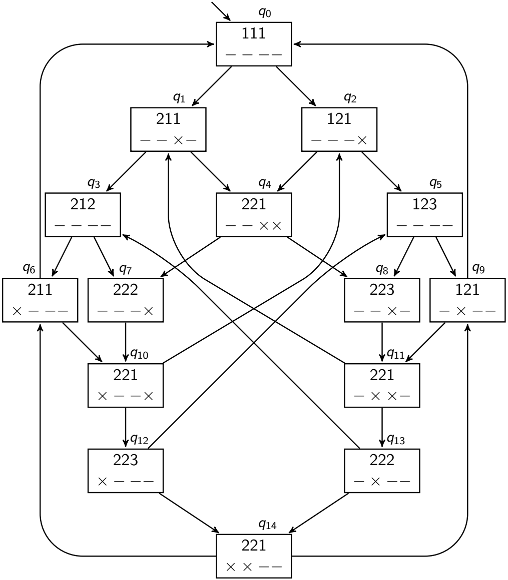
A more realistic example!

1 server, 2 clients, 4 channels

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

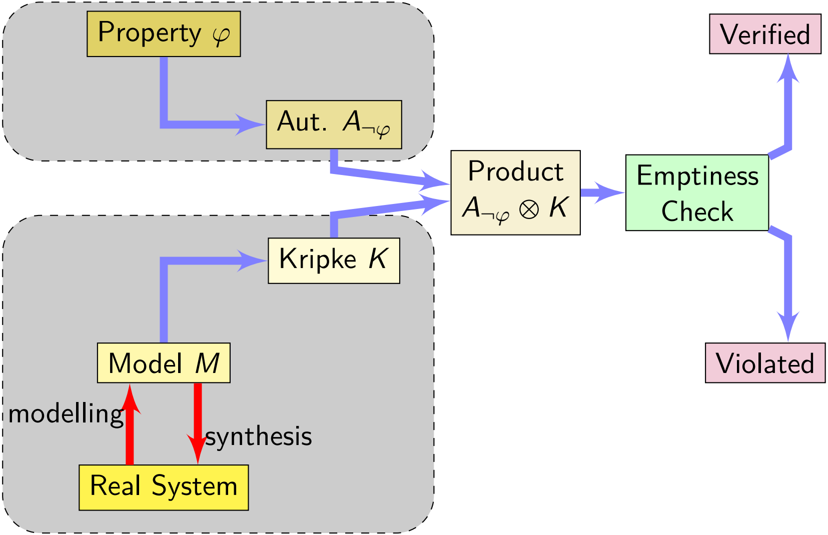
$$
\begin{array}{ll}
(1) & \langle\ s\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ a\ ,\ .\ \rangle \\
(2) & \langle\ .\ ,\ s\ ,\ .\ ,\ .\ ,\ .\ ,\ .\ ,\ a\ \rangle \\
(3) & \langle\ r\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ ,\ .\ ,\ .\ \rangle \\
(4) & \langle\ .\ ,\ r\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ ,\ .\ \rangle \\
(5) & \langle\ .\ ,\ .\ ,\ r_1\ ,\ .\ ,\ .\ ,\ d\ ,\ .\ \rangle \\
(6) & \langle\ .\ ,\ .\ ,\ s_1\ ,\ a\ ,\ .\ ,\ .\ ,\ .\ \rangle \\
(7) & \langle\ .\ ,\ .\ ,\ r_2\ ,\ .\ ,\ .\ ,\ .\ ,\ d\ \rangle \\
(8) & \langle\ .\ ,\ .\ ,\ s_2\ ,\ .\ ,\ a\ ,\ .\ ,\ .\ \rangle
\end{array}
$$

Example's state space



Converting the state space in something usable

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

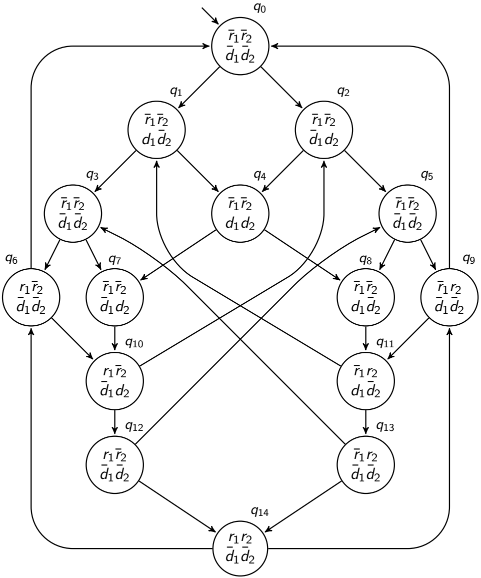
- AP is the set of atomic propositions
- $\mathcal{Q}$ is the finite set of state
- $q^0 \in \mathcal{Q}$ is the initial state
- $\delta : \mathcal{Q} \mapsto 2^{\mathcal{Q}}$ is the transition function that associates successors to a given state
- $l : \mathcal{Q} \mapsto 2^{AP}$ is labelling function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

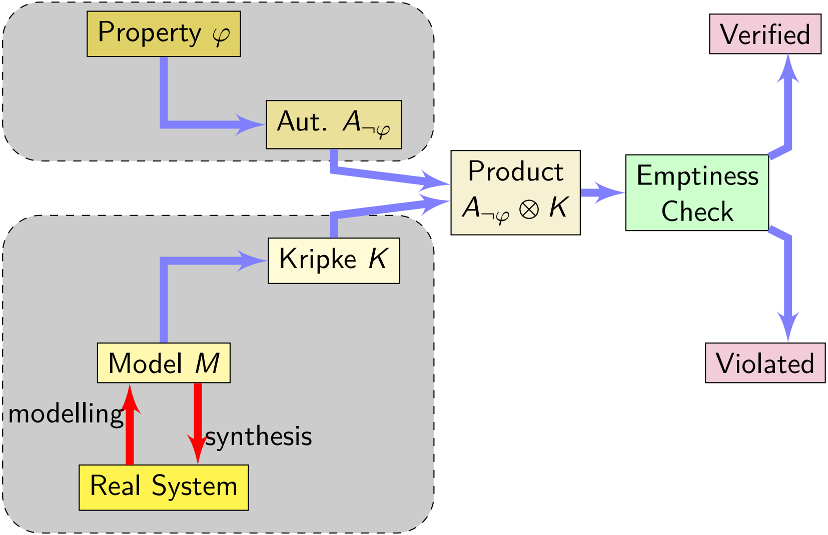
- r_1: a response is in progress between the server and the first client
- r_2: a response is in progress between the server and the second client
- d_1: a request (d for demand) is in progress between the first client and the server
- d_2: a request (d) is in progress between the second client and the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

Kripke Structure for the example



Product and Emptiness-check

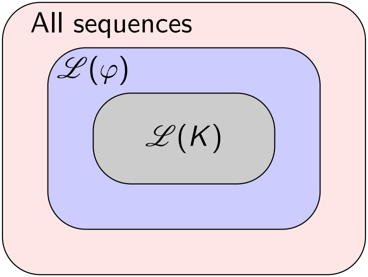
Automata approach for model checking



Why to check $\mathscr{L}(A_{\neg\varphi} \otimes K) \stackrel{?}{=} \emptyset$? (1/2)

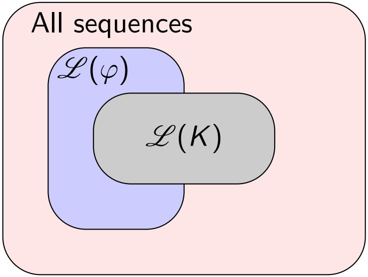
We want to check $\mathscr{L}(K) \subseteq \mathscr{L}(\varphi)$

$\mathscr{L}(K) \subseteq \mathscr{L}(\varphi)$



Property Verified

$\mathscr{L}(K) \not\subseteq \mathscr{L}(\varphi)$



Property Violated

Why to check $\mathscr{L}(A_{\neg\varphi} \otimes K) \stackrel{?}{=} \emptyset$? (2/2)

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

Emptiness check

Find if an accepting run exist.

Emptiness Checks

	degeneralized (**one** acceptance condition)	generalized (m acceptance conditions)
Nested DFS	DFS for acc. transitions + Nested DFS to find cycle ▪ 2 bits per state ▪ immediate counterexamples ▪ require degeneralization (size $\times m$) ▪ slow	DFS for acc. transitions + **several** Nested DFS ▪ $\log_2(m+1)$ bits per state ▪ visit states several times ▪ slow
SCC	(pointless)	Compute SCCs on the fly, abort on accepting SCC ▪ fastest ▪ visit states once ▪ indifferent to m ▪ SCC information useful ▪ one integer per state

Full example

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

$$'(\mathrm{G}(d_1 \text{ -> } \mathrm{F}\ r_1))'$$

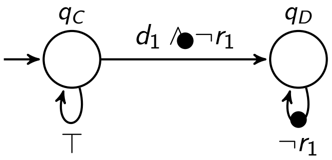
Express Property Automaton

How to express ?

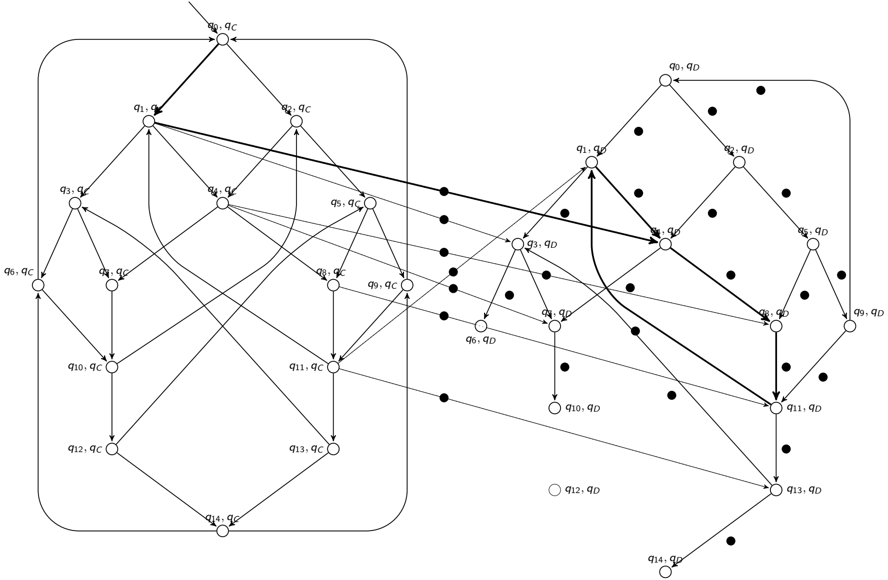
If client 1 send a request, he will necessarily receive a response

$$'(G(d_1 \text{ -> } F\ r_1))'$$

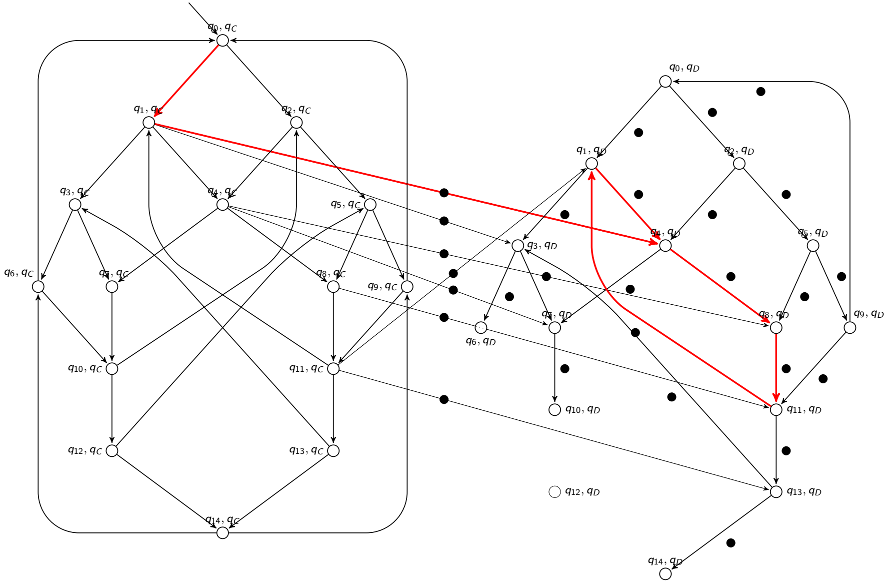
We can translate '!(G(d_1 -> F r_1))' into an automaton:



Product Kripke structure / Automaton



Emptiness check



Demo Time.