Logic, automata and model checking for discrete and hybrid systems

Tiziano Villa\textsuperscript{1} and Davide Bresolin\textsuperscript{2}

\textsuperscript{1}University of Verona, Italy
\textsuperscript{2}University of Padova, Italy

Hybrid systems: Computation and Control
La Sapienza Rome University
September 22, 2018
Outline

1. Temporal logic and property specification
2. Model checking on finite automata
3. Introduction to hybrid automata
4. Tools for the verification of hybrid automata
5. Conclusions
The Problem of Verification

- A design is correct when it meets its specification in the operating environment: "A design without a specification cannot be right or wrong, it can only be surprising!" (quoted from Young et al., 1985)
- Simulation of a number of tests is not sufficient to guarantee correctness especially for safety-critical applications (medical, robotics, avionics, automotive, ...)
- Since specifications or requirements of a design must be stated in a formal language to avoid ambiguities, mathematical specifications of system properties are introduced
Temporal logic is a mathematical formalism to specify system properties and it has been studied by philosophers and logicians since the times of Aristotle.

There are many types of temporal logic: linear temporal logic (LTL), computational tree logic (CTL, CTL*), etc.

In this lecture we will introduce linear temporal logic (LTL) by which one can express properties over a single but arbitrary execution of a system (Amir Pnueli, 1941-2009, Weizmann Institute of Science (Israel), was awarded the ACM Turing Award in 1996 for introducing temporal logic into computer science and for outstanding contributions to program and system verification).
Properties in Temporal Logic

- **Occurrence of an event and its properties.** E.g., An event $A$ must occur at least once in every trace of a system, or it must occur infinitely many times.
- **Causal dependency between events.** E.g., If an event $A$ occurs in a trace, then event $B$ must occur too.
- **Ordering of events.** E.g., Every occurrence of event $A$ is preceded by a matching occurrence of event $B$. 
An execution trace of a finite state machine (FSM) is a sequence of the form

\[ q_0, q_1, q_2, q_3, \ldots \]

where \( q_j = (x_j, s_j, y_j) \), with \( s_j \) is the state, \( x_j \) is the input valuation, \( y_j \) is the output valuation at reaction \( j \).

An atomic proposition is a statement (predicate, i.e., expression that evaluates to true or false) about the inputs, outputs or states of an FSM at the current time. E.g.,

- \( \text{true} \) True if input \( x \) is present
- \( x \) True if input \( x \) is present
- \( x = \text{present} \) True if input \( x \) is present
- \( x = 1 \) True if input \( x \) is present and has value 1
- \( y = \text{absent} \) True if output \( y \) is absent
- \( s \) True if FSM is in state \( s \)
A propositional logic formula is a predicate that combines atomic proposition using the standard logical connectives. It represents a more complex statement about an input, output, or state of an FSM at the current time. E.g.,

- $x \land y$ True if $x$ and $y$ are both present
- $x = \text{present} \land y = \text{absent}$ True if $x$ is present and $y$ is absent
- $\neg y$ True if $y$ is absent
- $a \rightarrow y$ True if when the FSM is in state $a$, the output $y$ will be made present by the reaction
Propositional formulas on traces

A propositional formula applies to an entire trace $q_0, q_1, q_2, \ldots$ and it holds for a trace iff it holds for the first element $q_0$ of the trace. Compare with propositional formulas which apply to just one reaction $q_i$. E.g., the propositional formula $\phi = p$ holds for the trace $q_0, q_1, q_2, \ldots$ iff $p$ is true for the first element $q_0$ of the trace.

Propositional formulas on FSMs

Given an FSM $M$ and a propositional formula $\phi$, we say that $\phi$ holds for $M$ iff $\phi$ holds for all traces of $M$. This requires considering all inputs.
Propositional formulas are the simplest LTL formulas. They apply to an entire trace rather than just to a single element of a trace, with the criterion that they hold for a trace iff they hold for the first element of that trace.

This is not enough: we need ways to reason about the entire trace and so we introduce some temporal operators which define general LTL formulas.

The temporal operators are:

- $G\phi$
- $F\phi$
- $X\phi$
- $\phi_1 U \phi_2$

This is Temporal LTL (PLTL); there is also first-order LTL with quantifiers $\forall, \exists$
LTL Formulas on Traces and FSMs

- **LTL formulas** are statements about an execution trace \( q_0, q_1, q_2, \ldots \), with the following meaning (\( p \) is a propositional logic formula and \( \phi \) is either a propositional logic formula or an LTL formula):
  
  1. \( \neg \phi = p \) holds iff \( p \) holds in \( q_0 \) (i.e., iff \( p \) is true for the first element \( q_0 \) of the trace)
  2. \( \neg \phi = G \phi \) holds iff \( \phi \) holds for every suffix of the trace
  3. \( \neg \phi = F \phi \) holds iff \( \phi \) holds for some suffix of the trace
  4. \( \neg \phi = X \phi \) holds iff \( \phi \) holds for the trace \( q_1, q_2, \ldots \)
  5. \( \phi = \phi_1 U \phi_2 \) holds iff \( \phi_1 \) holds for all suffixes of the trace until a suffix for which \( \phi_2 \) holds

- An LTL formula \( \tilde{\phi} \) holds for an **FSM** \( M \) iff \( \tilde{\phi} \) holds for all traces of \( M \).
FSMs for Running Example

**input:** $x$: pure

**output:** $y$: pure

Examples of LTL Propositional Formulas

- The LTL formula $a$ holds for the running FSM ($b$) because all traces begin in state $a$. The formula does not hold for the running FSM ($a$).

- The LTL formula $x \rightarrow y$ holds for both FSMs. In both cases, in the first reaction if $x$ is present then $y$ is present too.

- The LTL formula $y$ is false for both FSMs. In both cases, a counterexample is a trace where $x$ is absent in the first reaction.
Temporal logic Model checking Hybrid automata Tools Conclusions

Globally Operator

**G Operator**

- **Gφ (or □φ) (globally φ)** holds for a trace iff φ holds for every suffix of that trace, i.e., iff for all $j \geq 0$ formula φ holds for the suffix $q_j, q_{j+1}, q_{j+2}, \ldots$ (i.e., $p$ must hold for the traces $q_0, q_1, q_2, q_3, \ldots; q_1, q_2, q_3, \ldots; q_2, q_3, \ldots; q_3, \ldots$, etc.).

- If φ is a propositional logic formula, $G\phi$ simply means that φ holds in every reaction and it is also termed an invariant.
Temporal logic Model checking Hybrid automata Tools Conclusions

G Operator

Examples with $G$ operator

- The LTL formula $G(x \rightarrow y)$ holds for the FSM ($b$) of the running example.
- The LTL formula $G(x \land y)$ does not hold for the same FSM, because it is false for any trace where $x$ is absent in any reaction.
Eventually Operator

$F \varphi$ (or $\Diamond \varphi$) (eventually (finally, future) $\varphi$) holds for a trace iff $\varphi$ holds for some suffix of that trace, i.e., iff for some $j \geq 0$ formula $\varphi$ holds for the suffix $q_j, q_{j+1}, q_{j+2}, \ldots$.

If $\varphi$ is a propositional logic formula (e.g., $\varphi = p$), $F \varphi$ simply means that $\varphi$ holds for some $q_j$ (e.g., $p$ is true for $q_j$).
F Operator

Examples with F operator

- The LTL formula $Fb$ holds for the FSM ($a$) of the running example, because the FSM starts in state $b$, hence for all traces the proposition $b$ holds for the trace itself (the very first suffix).
- The LTL formula $G(x \rightarrow Fb)$ holds for the same FSM, because it $x$ is present in any reaction then the FSM will eventually be in state $b$ (true also for suffixes that start in state $a$).
- The LTL formula $(Gx) \rightarrow (Fb)$ holds trivially for the same FSM, because $Fb$ is true for all traces since the initial state is $b$. 
Next Operator

X Operator

- $X\phi$ (or $\bigcirc\phi$) (next state $\phi$) holds for a trace $q_0, q_1, q_2, q_3, \ldots$ iff $\phi$ holds for the trace $q_1, q_2, q_3, \ldots$.

- If $\phi$ is a propositional logic formula (e.g., $\phi = p$), $X\phi$ simply means that $\phi$ holds for $q_1$ (e.g., $p$ is true for $q_1$).
X Operator

Examples with X operator

- The LTL formula $x \rightarrow Xa$ holds for the FSM (a) of the running example, because if $x$ is present in the first reaction, then the next state will be a.

- The LTL formula $G(x \rightarrow Xa)$ does not hold for the same FSM, because it does not hold for any suffix that begins in state a.

- The LTL formula $G(b \rightarrow Xa)$ holds for the FSM (b).
Until Operator

U Operator

- $\phi_1 U \phi_2$ ($\phi_1$ until $\phi_2$) holds for a trace $q_0, q_1, q_2, q_3, \ldots$ iff $\phi_2$ holds for some suffix of that trace, and $\phi_1$ holds until $\phi_2$ becomes true, i.e., iff there is $j \geq 0$ such that $\phi_2$ holds for the suffix $q_j, q_{j+1}, q_{j+2}, \ldots$, and for all $i$ such that $0 \leq i < j$ formula $\phi_1$ holds for the suffixes $q_i, q_{i+1}, q_{i+2}, \ldots$ ($\phi_1$ may or may not hold for $q_j, q_{j+1}, q_{j+2}, \ldots$).

- A weaker form of Until does not require $\phi_2$ to hold eventually and it is equivalent to $(G \phi_1) \lor (\phi_1 U \phi_2)$, i.e., either $\phi_1$ always holds for any suffix, or if $\phi_2$ holds for some suffix then $\phi_1$ holds for all previous suffixes: $(F \neg \phi_1) \rightarrow (\phi_1 U \phi_2)$. 
Until Operator

Examples with $U$ operator

- The LTL formula $aUx$ is false for the FSM (b) in the running example. The reason is that it holds only for the traces where $Fx$ holds, but the latter traces do not include all traces (they do not include the traces where $G\neg x$ holds, i.e., the traces where $q = (\neg x, a, \neg y)$ is repeated forever).

- It follows that the LTL formula $(G\neg x) \lor (aUx)$ holds for the FSM (b).
## Examples of LTL Formulas

### Basic operators:
- \( Gp \): \( p \) holds in all state
- \( Fp \): \( p \) holds eventually
- \( Xp \): \( p \) holds in the next state

### Important patterns:
- \( GFp \): \( p \) holds infinitely often
- \( FGp \): eventually, \( p \) holds henceforth
- \( G(p \rightarrow Fq) \): every \( p \) is eventually followed by a \( q \)
- \( F(p \rightarrow (XXq)) \): if \( p \) occurs, then on some occurrence it is followed by a \( q \) two reactions later
Relations between Temporal Operators

**U Operator**
- Can one express $G\phi$ in terms of $F, p$ and propositional connectives? Yes: $G\phi = \neg F \neg \phi$.
- Can one express $F\phi$ in terms of $U$? Yes: $F\phi = true \ U\phi$.
- Can one express $X$ in terms of $G, F$ or $U$? No.
Safety and Liveness Properties

Intuition
Properties can be classified as safety or liveness properties. Safety properties specify that nothing bad happens, liveness properties specify that something good will eventually happen.

Definition
- $p$ is a safety property iff for any trace violating the safety property $p$ there is a finite-length (bad) prefix of the trace that cannot be extended to an infinite trace satisfying $p$. Finite-length error trace.
- $p$ is a liveness property iff every finite-length trace can be extended by a (good) suffix to an infinite trace that satisfies $p$. Infinite-length error trace.
Temporal logic Model checking Hybrid automata Tools Conclusions

Safety and Liveness: Examples

- Invariants like $Gp$ are safety properties.
- Liveness properties specify progress requirements; e.g., $F\phi$ is a liveness property: no finite trace can prove that it is not satisfied.
- $Xp$ safety.
- $GFP$ liveness.
- $G(p \rightarrow Fq)$ liveness.
- $G(p \rightarrow Xq)$ safety.
- $pUq$ both (Alpern and Schneider, 1985: Every property is the intersection of a safety property and a liveness property).
LTL in the Industrial Practice

There are specification languages for industrial applications based on LTL, e.g.:

1. PSL (Property Specification Language), an IEEE standard
2. PSL/Sugar (variant by IBM)

Examples:

1. \[ \text{assert always req } \rightarrow \text{next (ack until grant); equivalent to } G(r \rightarrow X(aUg)). \]
2. \[ \text{assert always req } \rightarrow \text{next[3] (grant); equivalent to } G(r \rightarrow XXXg). \]
Computational Tree Logic

- **Computational Tree Logic (CTL*)** provides explicitly quantifiers over traces of an FSM. E.g., we can write a $CTL^*$ expression holding for an FSM if there is any trace that satisfies some property, rather than requiring as in LTL that the property holds for all traces.

- It is called also **branching-time logic** because, when a reaction has a nondeterministic choice, it will consider simultaneously all options, whereas LTL (linear-time logic) considers only one trace at a time.

- The semantics that an LTL formula holds for an FSM if it holds for all traces cannot be expressed directly in LTL because LTL does not quantify over traces (we cannot say “$\forall$ traces etc.”), but it can expressed directly in $CTL^*$. 
Outline

1. Temporal logic and property specification
2. Model checking on finite automata
3. Introduction to hybrid automata
4. Tools for the verification of hybrid automata
5. Conclusions
where $S$ and $E$ are represented as automata.
Reachability analysis and model checking

Reachability analysis
The process of computing the set of reachable states for a system.
Reachability analysis and model checking

Reachability analysis
The process of computing the set of reachable states for a system.

Model checking
An algorithmic method for determining whether a system satisfies a formal specification expressed in temporal logic.
Reachability analysis and model checking

**Reachability analysis**
The process of computing the set of reachable states for a system.

**Model checking**
An algorithmic method for determining whether a system satisfies a formal specification expressed in temporal logic.

Model checking typically performs reachability analysis.
Consider an LTL formula of the form $G \ p$ where $p$ is a proposition ($p$ is a property on a single state)
Consider an LTL formula of the form $G \ p$ where $p$ is a proposition ($p$ is a property on a single state)

**Approach**

To verify $G \ p$ on a system $M$, one simply needs to enumerate all the reachable states and check that they all satisfy $p$. 
Traffic light controller example

Property $p$: $G(\neg (\text{green} \land \text{crossing}))$

Input: $\text{sigR, sigG, sigY}$: pure
Output: $\text{pedestrian}$: pure

$M$
Property $p$: $G(\neg (\text{green} \land \text{crossing}))$

Variable: $\text{count}: \{0, \cdots, 60\}$

- $\text{count} = \text{count} + 1$
- $\text{count} = 0$
- $\text{count} \geq 60 / \text{count} := 0$
- $\text{count} < 60 / \text{count} := \text{count} + 1$

Diagram:
- States: green, none, red, crossing, yellow, waiting, pending, waiting
- Transitions:
  - $\text{count} = \text{count} + 1$
  - $\text{count} = 0$
  - $\text{count} \geq 60 / \text{count} := 0$
  - $\text{count} < 60 / \text{count} := \text{count} + 1$
Number of states

The state space found is typically represented as a directed graph called a state graph.
The state space found is typically represented as a directed graph called a state graph.

When $M$ is a finite-state machine, this reachability analysis will terminate (in theory). In practice, the number of states may be very large consuming too much run-time or memory (the state explosion problem).
Property $p$: $G(¬(\text{green} \land \text{crossing}))$

**variable:** $\text{count}$: $\{0, \ldots, 60\}$

The example has 189 states, considering all possible values of $\text{count}$:

$$R = (\text{red, crossing, 0}), (\text{red, crossing, 1}), \ldots (\text{red, crossing, 60}), (\text{green, none, 0}), (\text{green, none, 1}), \ldots, (\text{green, none, 60}), (\text{yellow, waiting, 0}), \ldots (\text{yellow, waiting, 5}), (\text{pending, waiting, 1}), \ldots, (\text{pending, waiting, 60})$$
The symbolic approach

Rather than exploring new reachable states one at a time, we can explore new sets of reachable states.

- However, we only represent sets implicitly, as Boolean functions.
The symbolic approach

Rather than exploring new reachable states one at a time, we can explore new sets of reachable states.

- However, we only represent sets implicitly, as Boolean functions.

Set operations can be performed using Boolean algebra. Represent a finite set of states $S$ by its characteristic Boolean function $f_S$:

$$f_S(x) = 1 \text{ iff } x \in S$$

Similarly, the state transition function $\delta$ yields a set $\delta(s)$ of next states from current state $s$, and so can also be represented using a characteristic Boolean function for each $s$. 
The symbolic reachability algorithm

■ **Input:** initial state $s_0$ and transition relation $\delta$ for closed finite-state system $M$, represented symbolically

■ **Output:** set $R$ of reachable states of $M$, represented symbolically

1. **Initialize:** current set of reached states $R = \{s_0\}$
2. **Symbolic Search()** {
3. $R_{new} = R$
4. **while** $R_{new} \neq 0$ **do**
5. $R_{new} := \{s' | \exists s \in R \text{ s.t. } s' \in \delta(s)\} \setminus R$
6. $R := R \cup R_{new}$
7. **end**
8. **}
Symbolic model checking example

Property $p$: \( G(\neg (green \land crossing)) \)

**variable:** count: \( \{0, \ldots, 60\} \)

\[
R = \begin{cases} 
(v_1 = red \land v_2 = crossing \land 0 \leq count \leq 60) \lor \\
(v_1 = green \land v_2 = none \land 0 \leq count \leq 60) \lor \\
(v_1 = pending \land v_2 = waiting \land 0 \leq count \leq 60) \lor \\
(v_1 = yellow \land v_2 = waiting \land 0 \leq count \leq 5)
\end{cases}
\]
Abstraction in model checking

Should use simplest model of a system that provides proof of safety

- On one hand, they provide *more* behaviors than the original model
- On the other, they have *smaller* state spaces, hence they are easier to check
Abstraction in model checking

Should use simplest model of a system that provides proof of safety

- On one hand, they provide more behaviors than the original model
- On the other, they have smaller state spaces, hence they are easier to check

The challenge is to know what details can be abstracted away

A simple and useful approach is called localization abstraction.

- It hides state variables that are irrelevant to the property being verified
Abstract model for traffic light example

Property $p$: $G(\neg(\text{green} \land \text{crossing}))$
Suppose we have a robot that must pick up multiple things, in any order.

\[ \phi_i = \text{robot picks up item } i, \text{ where } 1 \leq i \leq n \]

Goal to be achieved is

\[ F\phi_1 \land F\phi_2 \land \cdots \land F\phi_n \]
Now suppose we have a robot that must pick up multiple things, in a specified order.

\( \phi_i = \text{robot picks up item } i, \text{ where } 1 \leq i \leq n \)

Goal to be achieved is

\[ F(\phi_1 \land F(\phi_2 \land \cdots \land F\phi_n)) \]
Now suppose we have a robot that must pick up multiple things, in a specified order.

\( \phi_i = \text{robot picks up item } i, \text{ where } 1 \leq i \leq n \)

Goal to be achieved is

\[ F(\phi_1 \land F(\phi_2 \land \cdots \land F\phi_n)) \]

How can we use model checking to synthesize a control strategy?
Let’s consider the first part alone: \( F(\phi_1) \)

Recall that

\[
F(\phi_1) = \neg G(\neg \phi_1)
\]
Let’s consider the first part alone: $F(\phi_1)$

Recall that

$$F(\phi_1) = \neg G(\neg \phi_1)$$

Therefore we can construct a counterexample to $G(\neg \phi_1)$

The counterexample is given by a trace that gets the robot to the desired point.
An example

A robot delivery service, with moving obstacles
\( \phi = \text{robot destination} \)
An example

A robot delivery service, with moving obstacles
\( \phi = \text{robot destination} \)

At any time step:

- Robot \( R \) can move left, right, up, down, stay put
- Environment \( E \) can move one obstacle up or down or stay put, but only three times total
An example

A robot delivery service, with moving obstacles
\( \phi = \text{robot destination} \)

At any time step:
- Robot \( R \) can move left, right, up, down, stay put
- Environment \( E \) can move one obstacle up or down or stay put, but only three times total

Can model \( R \) and \( E \) as FSMs
- \( R \) state: its position;
- \( E \) state: positions of obstacles and count value.
An example

A robot delivery service, with moving obstacles
φ = robot destination
A robot delivery service, with moving obstacles

\( \phi = \text{robot destination} \)

Goal to be achieved can be stated in temporal logic: \( F\phi \)
An example

A robot delivery service, with moving obstacles
\( \phi = \text{robot destination} \)

Goal to be achieved can be stated in temporal logic: \( F\phi \)

How can we find a path for the robot from starting point to the destination?

- This is an example of a reachability problem.
Outline

1. Temporal logic and property specification
2. Model checking on finite automata
3. Introduction to hybrid automata
4. Tools for the verification of hybrid automata
5. Conclusions
Motivation

We will consider automata with an infinite number of states.

We will discuss the specification and analysis of systems involving variables either discrete or continuous.
Many real systems have a double nature. They:
- evolve in a **continuous** fashion
- are controlled by a **discrete** system

Such systems are called **hybrid systems** and may be modeled by **hybrid automata**
Example: 4-strokes engine

- **Intake stroke**: air and vaporized fuel are drawn in
- **Compression stroke**: fuel vapor and air are compressed and ignited
- **Combustion stroke**: fuel combusts and piston is pushed downwards
- **Exhaust/Emission stroke**: exhaust is driven out
- During 1st, 2nd and 4th stroke the piston is relying on the power and momentum generated by the pistons of the other cylinders

During the 4 strokes pression, temperature, . . . , vary continuously
The intuition

A hybrid automaton $H$ is a finite-state automaton with continuous variables $Z$.

A state is a couple $\langle v, r \rangle$ where $r$ is a valuation for $Z$. 
Syntax

Definition (Hybrid Automata (Piazza et al.))

A \( k \)-hybrid automaton \( H = \langle Z, Z', V, E, Inv, Dyn, Act, Reset \rangle \) consists of the following components:
Definition (Hybrid Automata (Piazza et al.))

A $k$-hybrid automaton $H = \langle Z, Z', \mathcal{V}, \mathcal{E}, \text{Inv}, \text{Dyn}, \text{Act}, \text{Reset} \rangle$ consists of the following components:

1. $Z = (Z_1, \ldots, Z_k)$ and $Z' = (Z'_1, \ldots, Z'_k)$ are two vectors of variables ranging over the reals;
Syntax

Definition (Hybrid Automata (Piazza et al.))

A \( k \)-hybrid automaton \( H = \langle Z, Z', \mathcal{V}, \mathcal{E}, \text{Inv}, \text{Dyn}, \text{Act}, \text{Reset} \rangle \) consists of the following components:

1. \( Z = (Z_1, \ldots, Z_k) \) and \( Z' = (Z'_1, \ldots, Z'_k) \) are two vectors of variables ranging over the reals;
2. \( \langle \mathcal{V}, \mathcal{E} \rangle \) is a finite directed graph;
Definition (Hybrid Automata (Piazza et al.))

A $k$-hybrid automaton $H = \langle Z, Z', V, E, Inv, Dyn, Act, Reset \rangle$ consists of the following components:

1. $Z = (Z_1, \ldots, Z_k)$ and $Z' = (Z'_1, \ldots, Z'_k)$ are two vectors of variables ranging over the reals;
2. $\langle V, E \rangle$ is a finite directed graph;
3. Each $v \in V$ is labeled by the two formulæ $Inv(v)[Z]$ and $Dyn(v)[Z, Z', T]$ such that if $Inv(v)[p]$ holds then $Dyn(v)[p, p, 0]$ holds as well;
Syntax

Definition (Hybrid Automata (Piazza et al.))

A \( k \)-hybrid automaton \( H = \langle Z, Z', \mathcal{V}, \mathcal{E}, \text{Inv}, \text{Dyn}, \text{Act}, \text{Reset} \rangle \) consists of the following components:

1. \( Z = (Z_1, \ldots, Z_k) \) and \( Z' = (Z'_1, \ldots, Z'_k) \) are two vectors of variables ranging over the reals;

2. \( \langle \mathcal{V}, \mathcal{E} \rangle \) is a finite directed graph;

3. Each \( \nu \in \mathcal{V} \) is labeled by the two formulæ \( \text{Inv}(\nu)[Z] \) and \( \text{Dyn}(\nu)[Z, Z', T] \) such that if \( \text{Inv}(\nu)[p] \) holds then \( \text{Dyn}(\nu)[p, p, 0] \) holds as well;

4. Each \( e \in \mathcal{E} \) is labeled by the formulæ \( \text{Act}(e)[Z] \) and \( \text{Reset}(e)[Z, Z'] \).
Comments on the definition

- \textit{Inv}, \textit{Dyn}, \textit{Act}, \textit{Reset} are sets of formulae in a first-order language $\mathcal{L}$
- E.g., $\mathcal{L} = (+, *, <, 0, 1)$
Comments on the definition

- \textit{Inv, Dyn, Act, Reset} are sets of formulae in a first-order language \( \mathcal{L} \)
  
  E.g., \( \mathcal{L} = (+, *, <, 0, 1) \)

- the formulae are evaluated over a model \( \mathcal{M} \) of \( \mathcal{L} \) in the domain \( \mathbb{R} \)
  
  E.g., \( \mathcal{M} = (\mathbb{R}, +, *, <, 0, 1) \)
Comments on the definition

- **Inv, Dyn, Act, Reset** are sets of formulae in a first-order language $\mathcal{L}$
  E.g., $\mathcal{L} = (+, \ast, <, 0, 1)$

- The formulae are evaluated over a model $\mathcal{M}$ of $\mathcal{L}$ in the domain $\mathbb{R}$
  E.g., $\mathcal{M} = (\mathbb{R}, +, \ast, <, 0, 1)$

- The nodes $\mathcal{V}$ are called locations (or control modes), the arcs $\mathcal{E}$ are called control switches
Comments on the definition

- \( \text{Inv, Dyn, Act, Reset} \) are sets of formulae in a first-order language \( \mathcal{L} \)
  
  E.g., \( \mathcal{L} = (+, *, <, 0, 1) \)

- The formulae are evaluated over a model \( \mathcal{M} \) of \( \mathcal{L} \) in the domain \( \mathbb{R} \)
  
  E.g., \( \mathcal{M} = (\mathbb{R}, +, *, <, 0, 1) \)

- The nodes \( \mathcal{V} \) are called locations (or control modes), the arcs \( \mathcal{E} \) are called control switches

- The variable \( T \) represents time
Comments on the definition

- \(\text{Inv}, \text{Dyn}, \text{Act}, \text{Reset}\) are sets of formulae in a first-order language \(\mathcal{L}\).
  
  E.g., \(\mathcal{L} = (+, *, <, 0, 1)\)

- The formulae are evaluated over a model \(\mathcal{M}\) of \(\mathcal{L}\) in the domain \(\mathbb{R}\).
  
  E.g., \(\mathcal{M} = (\mathbb{R}, +, *, <, 0, 1)\)

- The nodes \(\mathcal{V}\) are called locations (or control modes), the arcs \(\mathcal{E}\) are called control switches.

- The variable \(T\) represents time.

- \(p \in \mathbb{R}^k\)
Example: thermostat

Let us consider a room heated by a radiator controlled by a thermostat:

- When the thermostat is on the temperature increases exponentially in time.
- When the thermostat is off the temperature decreases exponentially in time.
- The thermostat switches on the radiator when the temperature decreases below 19°C.
- The thermostat switches off the radiator when the temperature increases above 21°C.
Example: thermostat

Let us model the behaviour of the temperature in time by an hybrid automaton $H$ with:

- 2 locations ON and OFF
- 2 arcs that join the two locations
- 1 continuous variable $Z$ that represents the temperature
Example: thermostat

\[ H = \langle Z, Z', \mathcal{V}, \mathcal{E}, \text{Inv}, \text{Dyn}, \text{Act}, \text{Reset}\rangle \text{ such that:} \]

- \( Z \) and \( Z' \) are two variables
- \( \mathcal{V} = \{\text{ON}, \text{OFF}\} \) and \( \mathcal{E} = \{(\text{ON}, \text{OFF}), (\text{OFF}, \text{ON})\} \)
- \( \text{Inv}(\text{ON})[Z] := Z \leq 22 \) and
  \[ \text{Dyn}(\text{ON})[Z, Z', T] := Z' = Z \times e^T \]
- \( \text{Inv}(\text{OFF})[Z] := Z \geq 18 \) and
  \[ \text{Dyn}(\text{OFF})[Z, Z', T] := Z' = Z / e^T \]
- \( \text{Act}((\text{ON}, \text{OFF}))[Z] := Z \geq 21 \) and
  \( \text{Reset}((\text{ON}, \text{OFF})[Z, Z'] := Z' = Z \)
- \( \text{Act}((\text{OFF}, \text{ON}))[Z] := Z \leq 19 \) and
  \( \text{Reset}((\text{OFF}, \text{ON}))[Z, Z'] := Z' = Z \)
Example: thermostat

\[ Z \geq 21 \]
\[ Z' = Z \]

\[ Z \leq 22 \]
\[ Z' = Z \cdot e^T \]

\[ Z \geq 18 \]
\[ Z' = Z \cdot e^{-T} \]

\[ Z \leq 19 \]
\[ Z' = Z \]
Semantics

\[ \ell = \langle v, r \rangle \text{ is admissible if } Inv(v)[r] \text{ holds} \]

Definition (Continuous transitions)

There exists a continuous function \( f : \mathbb{R}^+ \rightarrow \mathbb{R}^k \) such that \( r = f(0), \ s = f(t) \) and for each \( t' \in [0,t] \) the formulae \( Inv(v)[f(t')] \) and \( Dyn(v)[r, f(t'), t'] \) hold
Semantics

\[ \ell = \langle v, r \rangle \text{ is admissible if } Inv(v)[r] \text{ holds} \]

\[ \langle v, r \rangle \xrightarrow{\langle v', v' \rangle} D \langle v', s \rangle \iff \langle v, v' \rangle \in \mathcal{E}, \quad Inv(v)[r], \]
\[ Act(\langle v, v' \rangle)[r], \]
\[ Reset(\langle v, v' \rangle)[r, s] \]
\[ Inv(v')[s] \text{ hold} \] and
Reachability

\[ \langle v, r \rangle \xrightarrow{e} u \xleftarrow{e'} \langle u, s \rangle \xrightarrow{e} \omega \]

- **Temporal logic**
- **Model checking**
- **Hybrid automata**
- **Tools**
- **Conclusions**
Reachability
Reachability
Reachability
Reachability
Reachability
Let $I, F \in \mathbb{R}^k$. Can we reach $\langle u, F \rangle$ from $\langle v, I \rangle$?
Trace and reachability

A trace of $H$ is a sequence of admissible states $[\ell_0, \ell_1, \ldots, \ell_i, \ldots, \ell_n]$ such that $\ell_{i-1} \rightarrow \ell_i$ holds $\forall i \in [1, n]$.

**Definition (Reachability)**

The automaton $H$ reaches $\langle u, s \rangle$, $s \in \mathbb{R}^k$, from $\langle v, r \rangle$, $r \in \mathbb{R}^k$, if there exists a trace $tr = [\ell_0, \ldots, \ell_n]$ of $H$ such that $\ell_0 = \langle v, r \rangle$ and $\ell_n = \langle u, s \rangle$.

**Definition (Reachability problem)**

Given an automaton $H$, a set of starting points $\langle v, I \rangle$, $I \subseteq \mathbb{R}^k$, and a set of ending points $\langle u, F \rangle$, $F \subseteq \mathbb{R}^k$, decide whether there exists a point in $\langle v, I \rangle$ from which a point in $\langle u, F \rangle$ is reachable.
Many sources of non-determinism

Hybrid automata may be non-deterministic since:

- Different locations may partially share the invariants
- Different continuous trajectories may leave from the same admissible state
- There may be arcs that go to different locations but partially share the activation functions
- The activation functions are not necessarily on the frontiers of the invariants
- The reset functions are not necessarily deterministic
Many sources of non-determinism

Hybrid automata may be non-deterministic since:

- Different locations may partially share the invariants
Many sources of non-determinism

Hybrid automata may be non-deterministic since:

- Different locations may partially share the invariants
- Different continuous trajectories may leave from the same admissible state
Many sources of non-determinism

Hybrid automata may be non-deterministic since:

- Different locations may partially share the invariants
- Different continuous trajectories may leave from the same admissible state
- There may be arcs that go to different locations but partially share the activation functions
Many sources of non-determinism

Hybrid automata may be non-deterministic since:

- Different locations may partially share the invariants
- Different continuous trajectories may leave from the same admissible state
- There may be arcs that go to different locations but partially share the activation functions
- The activation functions are not necessarily on the frontiers of the invariants
Many sources of non-determinism

Hybrid automata may be non-deterministic since:

- Different locations may partially share the invariants
- Different continuous trajectories may leave from the same admissible state
- There may be arcs that go to different locations but partially share the activation functions
- The activation functions are not necessarily on the frontiers of the invariants
- The reset functions are not necessarily deterministic
Outline

1. Temporal logic and property specification
2. Model checking on finite automata
3. Introduction to hybrid automata
4. Tools for the verification of hybrid automata
5. Conclusions
Five families of tools

Tools for the verification of hybrid systems use different approaches:

1. Tools for exact verification:
   - Uppaal, HyTech

2. Tools that compute approximations of the reachable set:
   - PHAVer, SpaceEx, Flow*, HyPro/HyDRA, Ariadne

3. Tools that compute a discrete abstraction of the system:
   - HSOLVER, HybridSAL, HyCOMP

4. Tools that simulate the system:
   - Breach, S-TaLiRo, C2E2, HyLAA

5. Tools that apply automated theorem proving approaches:
   - Keymaera X, Isabelle/HOL
Exact verification of hybrid systems

- For systems with simple dynamics, the evolution can be computed exactly:
  - timed systems ($\dot{x} = 1$),
  - rectangular systems ($\dot{x} \in [c_{min}, c_{max}]$).

- Verification techniques for finite-state systems can be used

- The state space of the system can be computed

**PROS:** definite YES/NO answer, high performances

**CONS:** can verify only systems with simple dynamics
Uppaal

http://www.uppaal.org/

- developed by the Uppsala University, Sweden and the Aalborg University in Denmark
- written in Java and C++, graphical user interface
- limited to timed automata
- supports the composition of timed automata
- can verify complex properties in a subset of CTL temporal logic
- results are formally sound
- scalability: up to 100 continuous variables
http://embedded.eecs.berkeley.edu/research/hytech/

- developed at Cornell and Berkeley by Tom Henzinger, Pei-Hsin Ho, and Howard Wong-Toi
- written in C++, textual scripting interface
- one of the first tools for the verification of hybrid systems
- limited to rectangular automata
- supports the composition of automata
- results are formally sound
- computation is not guaranteed to terminate
- scalability: up to 10 continuous variables
Reachable set and safety properties

- The **reachable set** is the set of all states that can be reached by the system under dynamical evolution.
- It can be used to verify **safety properties**:
  - The system is safe if the reachable set is included in the safe set.
- The reachable set cannot be computed exactly (except for simple dynamics).
- Approximations can give positive and negative answers:
  - Overapproximations ⇒ positive answers
  - Underapproximations ⇒ negative answers

**PROS**: approximation can be very tight, can simulate the system

**CONS**: requires complex numerical analysis techniques
PHAVer

http://www-verimag.imag.fr/~frehse/phaver_web/

- developed at Verimag by Goran Frehse
- written in C++, textual scripting interface
- one of the first tools that enabled verification of hybrid automata
- affine dynamics and guards ($\dot{x} = Ax + b$)
- supports the composition of hybrid automata
- computes over-approximations of the reachable set
- state space is represented using polytopes
- results are formally sound (rational arithmetic with unlimited precision)
- scalability: up to 10 continuous variables
SpaceEx

http://spaceex.imag.fr/

- developed at Verimag by a team led by Oded Maler and Goran Frehse
- written in C++, graphical user interface
- affine dynamics and guards (\( \dot{x} = Ax + b \))
- supports the composition of hybrid automata
- computes over-approximations of the reachable set
- state space is represented using support functions and polytopes
- results are not guaranteed to be numerically sound (IEEE floating point arithmetic)
- one of the most scalable tools: up to a hundred continuous variables
Flow*

http://systems.cs.colorado.edu/research/cyberphysical/taylormodels

- developed at UC Boulder by a team led by Sankaranarayanan
- written in C++
- non-linear ODEs (polynomial dynamics inside modes, polyhedral guards on discrete transitions)
- computes over-approximations of the reachable set
- state space is represented by Taylor models (bounded degree polynomials over the initial conditions and time, bloated by an interval)
- results are guaranteed to be numerically sound (guaranteed interval arithmetic supported by MPFR)
- scalability: up to a dozen variables
HyPro/HyDRA

https://ths.rwth-aachen.de/research/projects/hypro/

- developed at RWTH Aachen University, Germany, by a team led by Erika Ábrahám
- C++ library for state set representations
- linear dynamics and guards
- different representations supported, with conversion between different representations
- templated number type
- fast implementation of specialized reachability analysis methods
- results are numerically sound only for some representations
ARIADNE

http://ariadne.parades.rm.cnr.it/

- developed by a joint team including the University of Maastricht, the University of Verona, and the company PARADES/ALES
- C++ library
- non-linear dynamics and guards
- supports the composition of hybrid automata
- computes both over- and lower- approximations of the reachable set
- state space is represented using Taylor image sets and kd-trees
- results are guaranteed to be sound (rigorous interval arithmetic)
- scalability: up to 10 continuous variables
An alternative approach, which **approximates the system** instead of the reachable set:

- compute a **discrete abstract system** that overapproximates the behavior of the original system
- does the abstract system **satisfy the property**?
  - **YES**: the original system is safe
  - **NO**: refine the abstraction and repeat the verification

**PROS**: can obtain a result in a few steps even for complex systems

**CONS**: no graphical output of the results, cannot simulate the system
HSOLVER

http://hsolver.sourceforge.net/

- developed by Stefan Ratschan
- written in C++, textual scripting interface
- non-linear dynamics and guards
- no support for the composition of hybrid automata
- uses constraint propagation techniques to approximate the system
- results are guaranteed to be sound (rational arithmetic with unlimited precision)
- scalability: up to 10 continuous variables
HybridSAL

http://sal.csl.sri.com/hybridsal/

- developed by Ashish Tiwari
- written in Java and LISP, textual scripting interface
- polynomial dynamics and guards
- supports the composition of hybrid automata
- uses predicate abstraction to abstract the discrete dynamics and qualitative reasoning to abstract the continuous dynamics
- results are guaranteed to be sound
- scalability: up to 10 continuous variables
HyCOMP

https://es.fbk.eu/tools/hycomp/

- developed at FBK by Alessandro Cimatti, Sergio Mover, Stefano Tonetta.
- written in C, textual scripting interface
- piece-wise linear and affine dynamics
- support the composition of hybrid automata
- solves different tasks: verification of invariant and LTL properties, verification of scenario specifications, parameter synthesis.
- verification algorithms based on Satisfiability Modulo Theory
- results are sound (infinite-precision arithmetic)
- scalability: up to 60 continuous variables
Falsification by simulation

- Explore the state space of the system by **computing a bunch of trajectories**
- If one of the trajectories violates the property, a **counterexample** is found
- If no counterexample is found, no conclusion can be made on the truth of the property
- Can be used to verify black-box systems

**PROS**: can manage complex properties and black-box systems

**CONS**: can only falsify the property
Breach

https://www.eecs.berkeley.edu/~donze/breach_page.html

- developed by Alexandre Donzé
- Matlab/C++ toolbox
- Systematic simulation is used to compute an underapproximation of the reachable set
- supports complex properties in Signal Temporal Logic
- supports parametric systems
S-TaLiRo

https://sites.google.com/a/asu.edu/s-taliro/

- developed by Sriram Sankaranarayanan
- Matlab toolbox
- uses a robustness metric to search for a counterexample
- randomized testing and stochastic optimization techniques are used to maximize the chance of finding the counterexample
- supports complex properties in Metric Temporal Logic
- supports parametric systems
C2E2

https://publish.illinois.edu/c2e2-tool/

- developed by the C2E2 development team at University of Illinois at Urbana-Champaign
- can verify bounded-time invariants of hybrid automata and Stateflow models
- it generates numerical simulations, and it iteratively refines reach set over-approximations to prove invariants
- it can also find counterexamples or bug traces
HyLAA

http://stanleybak.com/hylaa/

- developed by Stanley Bak with input from Parasara Sridhar Duggirala
- verification tool for system models with linear ODEs, time-varying inputs, and hybrid dynamics
- computes simulation-equivalent reachability: the set of states that can be reached by any fixed-step simulation under any possible input
- results are exact (under some restrictions)
- can generate counter-example traces when an error is found
Verification by theorem proving

- System under verification represented by a logical formula $Sys$
- Properties of interest represented by a logical formula $Prop$
- The verification problem is reduced to testing whether the formula $Sys \rightarrow Prop$ is valid (a logical tautology)
- Systems can be parametric and/or partially specified
- Very complex properties can be tested
- User intervention is needed to guide the proof

**PROS:** can manage complex properties and partially specified systems

**CONS:** semi-automatic approach
KeYmaera X

http://www.ls.cs.cmu.edu/KeYmaeraX/

- developed by André Platzer
- written in Java, graphical user interface
- hybrid systems are specified using a programming language
- non-linear dynamics and guards
- supports complex properties written in Differential Dynamic Logic
- pre-defined and custom tactics drive the automated proof search
Outline

1. Temporal logic and property specification
2. Model checking on finite automata
3. Introduction to hybrid automata
4. Tools for the verification of hybrid automata
5. Conclusions
Conclusions

- Formal Verification is a systematic way of checking whether all behaviors of a model of a system fulfill its specification.
- Formal Verification is in use in the industry in several different areas, mainly for the design of discrete hardware/software systems.
- Formal Verification can be applied to hybrid systems also, however, with limitations on the properties and/or on the dynamics of the system.
- Available tools use different approaches. The choice of the best one is strongly dependent on the application domain, on the system under verification.